Advanced Data Structures and Algorithm Analysis

丁尧相

浙江大学

Spring & Summer 2024

Lecture 1

2024-2-26

Outline:
Balanced Binary Search Trees (1)

- Binary search trees
- AVL trees
- Splay trees
- Amortized analysis
- Take-home messages

Acknowledgements:

This lecture is adapted from the slides designed by Prof. Yue Chen and the ZJU ADS course group.

Outline:
Balanced Binary Search Trees (1)

- Binary search trees
- AVL trees
- Splay trees
- Amortized analysis
- Take-home messages

Acknowledgements:

This lecture is adapted from the slides designed by Prof. Yue Chen and the ZJU ADS course group.

Data Structures

- Data structures represent dynamic sets of instances.
  - dynamic means the set can change.
  - can be ordered or unordered.

Data Structures

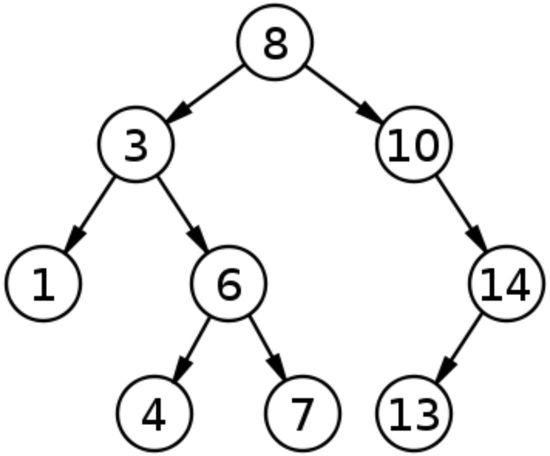
- Data structures represent dynamic sets of instances.
    - dynamic means the set can change.
    - can be ordered or unordered.
- Data structures are abstractions: supporting group of operations:
    - queries:
        - search, minimum, maximum, successor, predecessor…
    - modifying operations:
        - insert, delete…

Data Structures

- Data structures represent dynamic sets of instances.
  - dynamic means the set can change.
  - can be ordered or unordered.
- Data structures are abstractions: supporting group of operations:
  - queries:
    - search, minimum, maximum, successor, predecessor…
  - modifying operations:
    - insert, delete…
- A proper data structure effectively speeds up the set operations.

in terms of the size of the DS

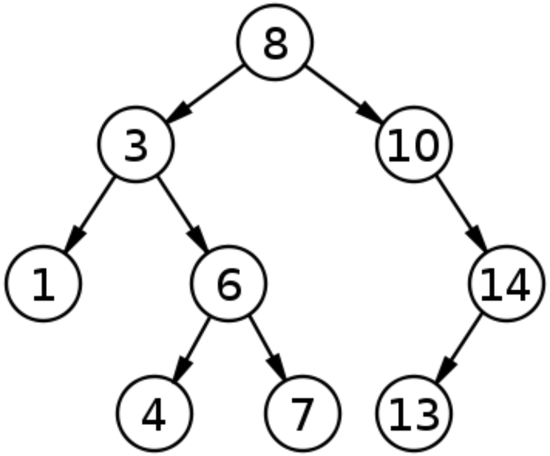
Binary Search Trees (BSTs)



- Every node has at most two children.

Figure courtesy: https://en.wikipedia.org/wiki/Binary_search_tree

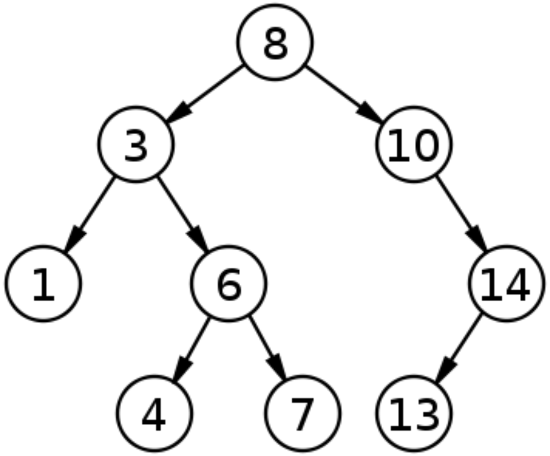
Binary Search Trees (BSTs)



- Every node has at most two children.
- The left child is smaller, and the right child is larger.

Figure courtesy: https://en.wikipedia.org/wiki/Binary_search_tree

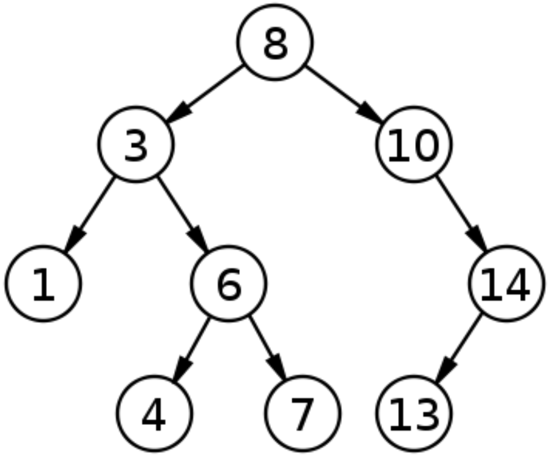
Binary Search Trees (BSTs)



- Every node has at most two children.
- The left child is smaller, and the right child is larger.
- The tree operations (search, insert, delete, minimum, maximum, successor, predecessor…) have time costs closely related to tree depth.

Figure courtesy: https://en.wikipedia.org/wiki/Binary_search_tree

Binary Search Trees (BSTs)

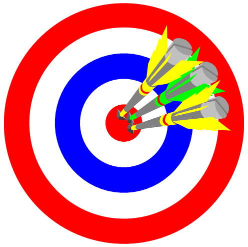


- Every node has at most two children.
- The left child is smaller, and the right child is larger.
- The tree operations (search, insert, delete, minimum, maximum, successor, predecessor…) have time costs closely related to tree depth.
- Balancing is to reduce tree depth in order to reduce time costs.

Figure courtesy: https://en.wikipedia.org/wiki/Binary_search_tree

Balanced BSTs

Balanced BSTs

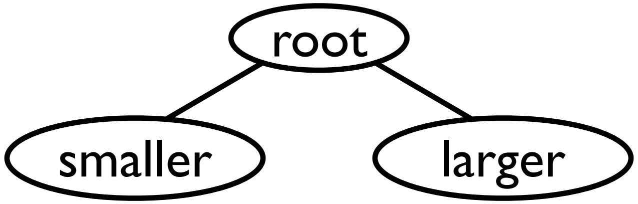


Target : Speed up searching (with insertion and deletion)

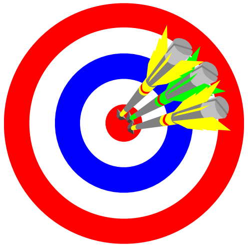
Balanced BSTs

Target : Speed up searching (with insertion and deletion)

Tool : Binary search trees

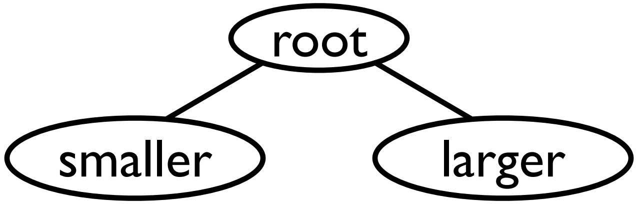


Balanced BSTs



Target : Speed up searching (with insertion and deletion)

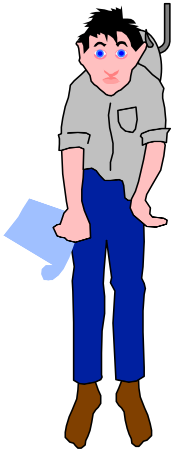
Tool : Binary search trees



root

smaller

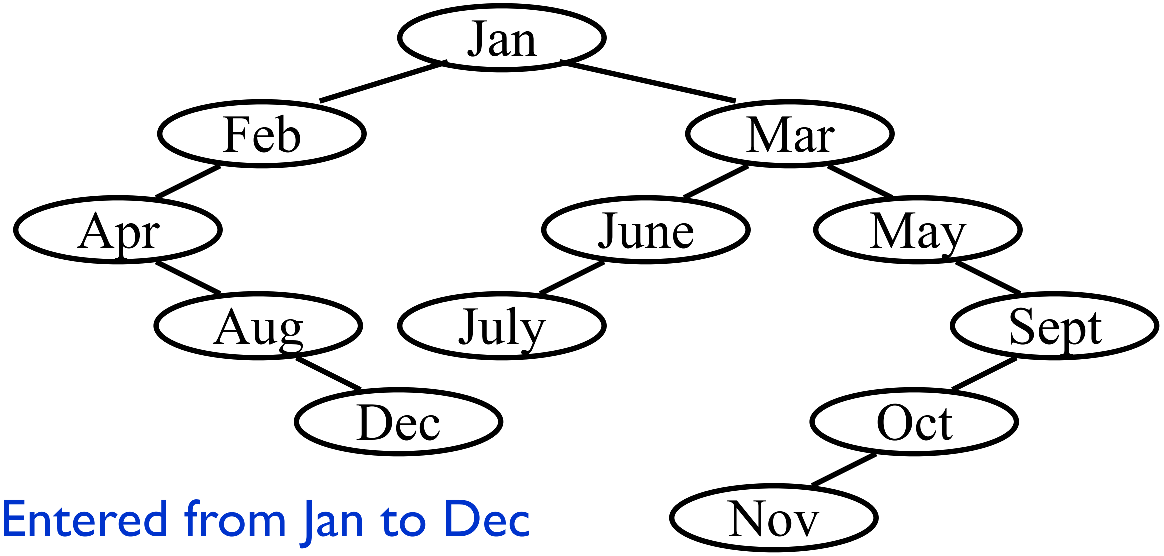
larger



Problem : Although T_p = O(height), but the height can be as bad as O(N).

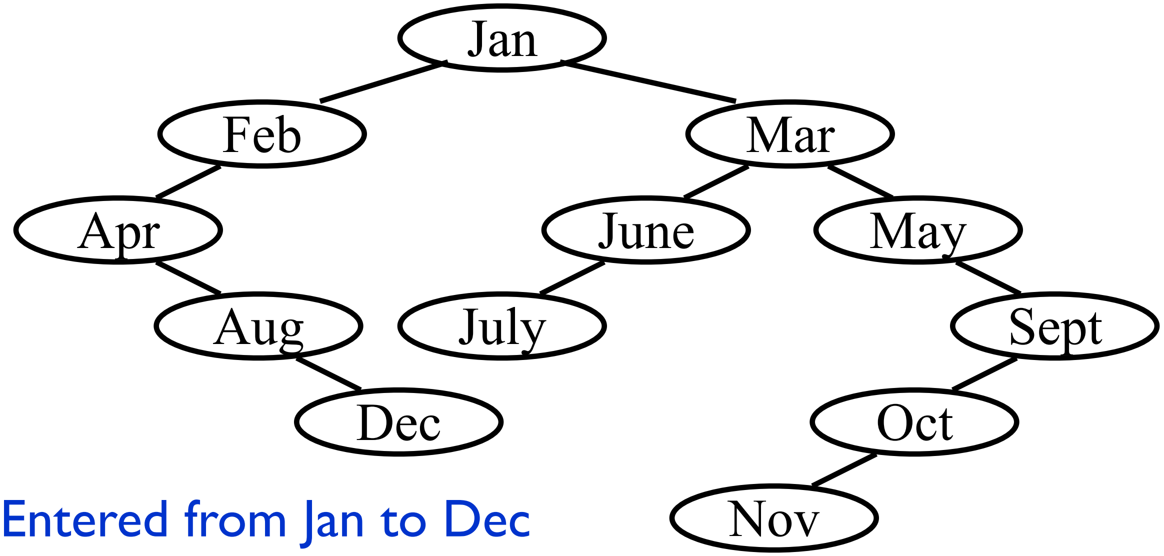
〖Example〗 2 binary search trees obtained for the months of the year

〖Example〗 2 binary search trees obtained for the months of the year

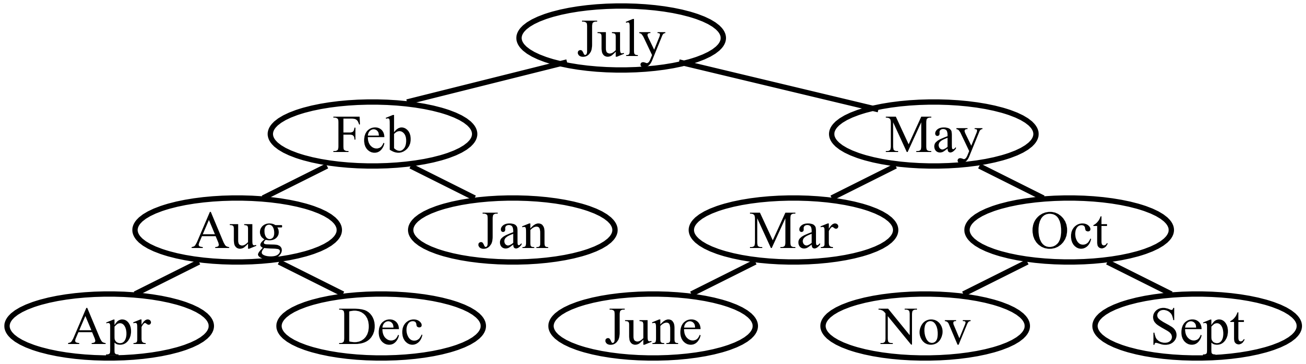


Entered from Jan to Dec

〚Example〛 2 binary search trees obtained for the months of the year

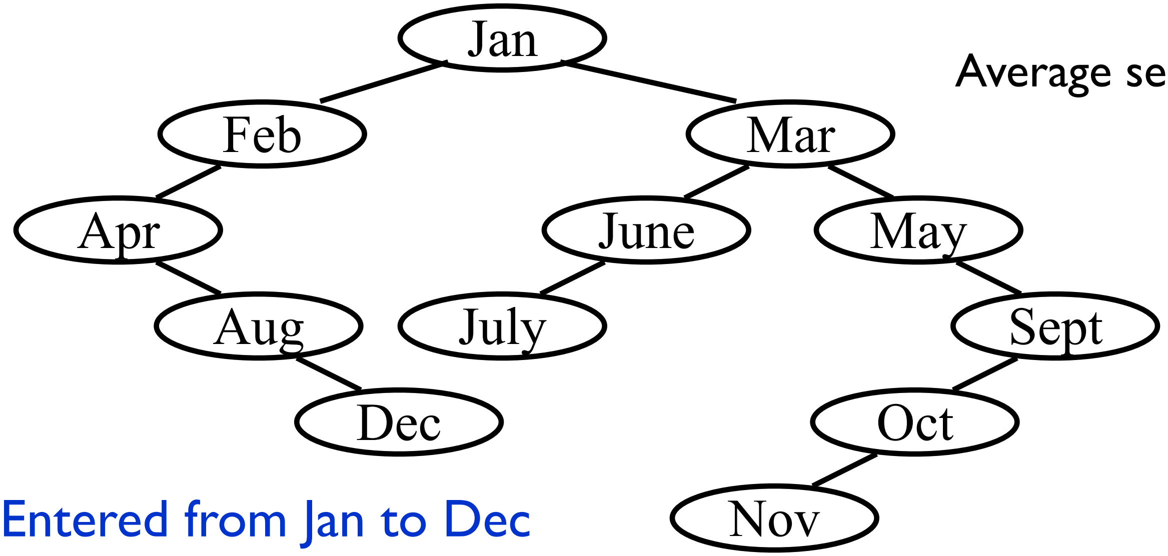


Entered from Jan to Dec



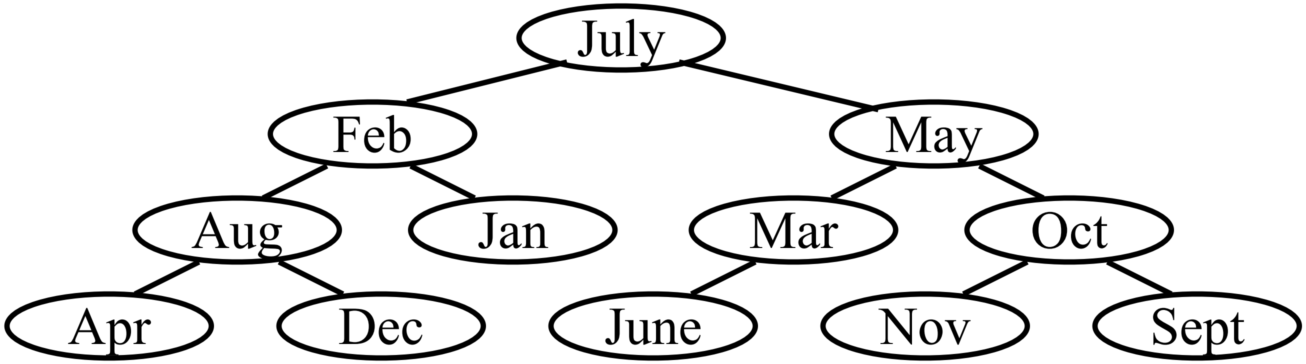
A balanced tree

〚Example〛 2 binary search trees obtained for the months of the year



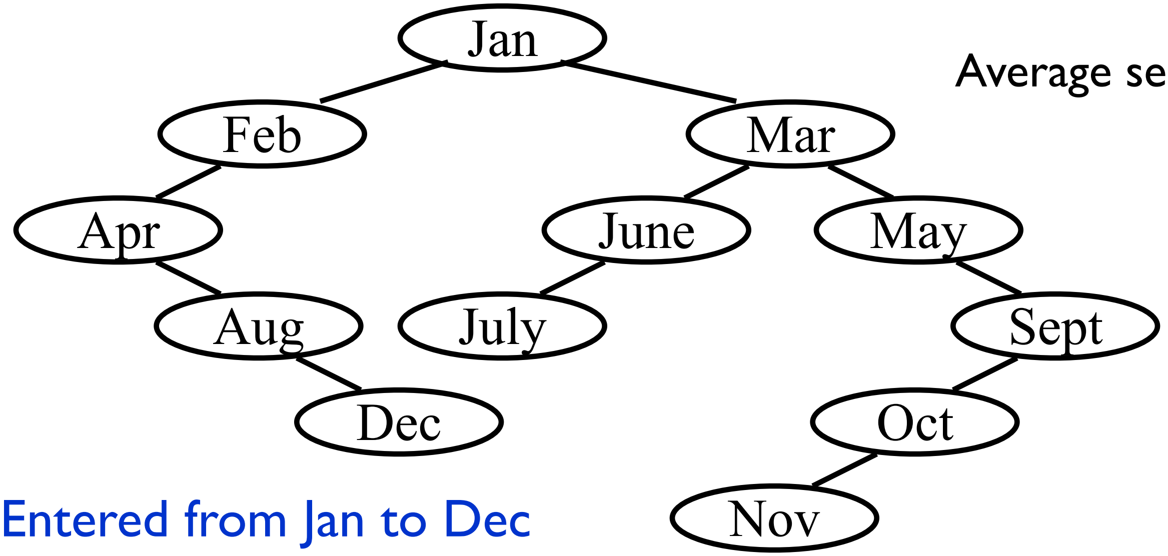
Average search time = 3.5

Entered from Jan to Dec



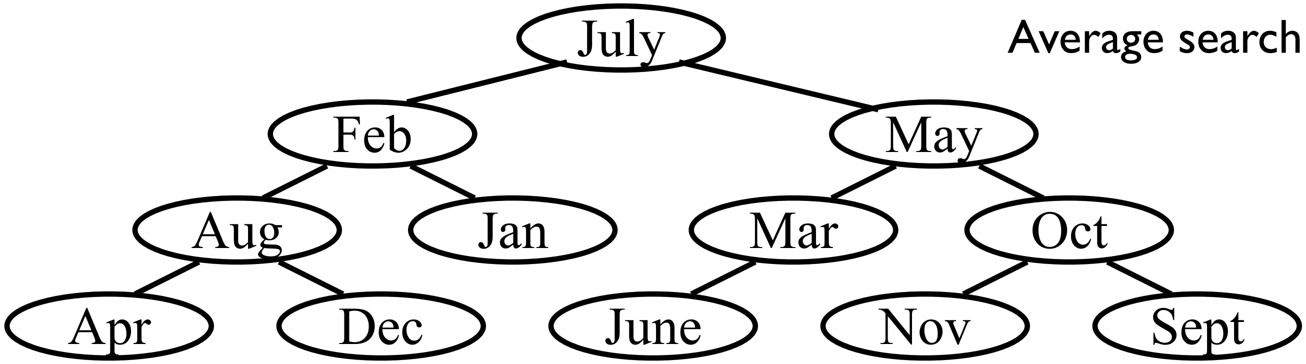
A balanced tree

〚Example〛 2 binary search trees obtained for the months of the year



Average search time = 3.5

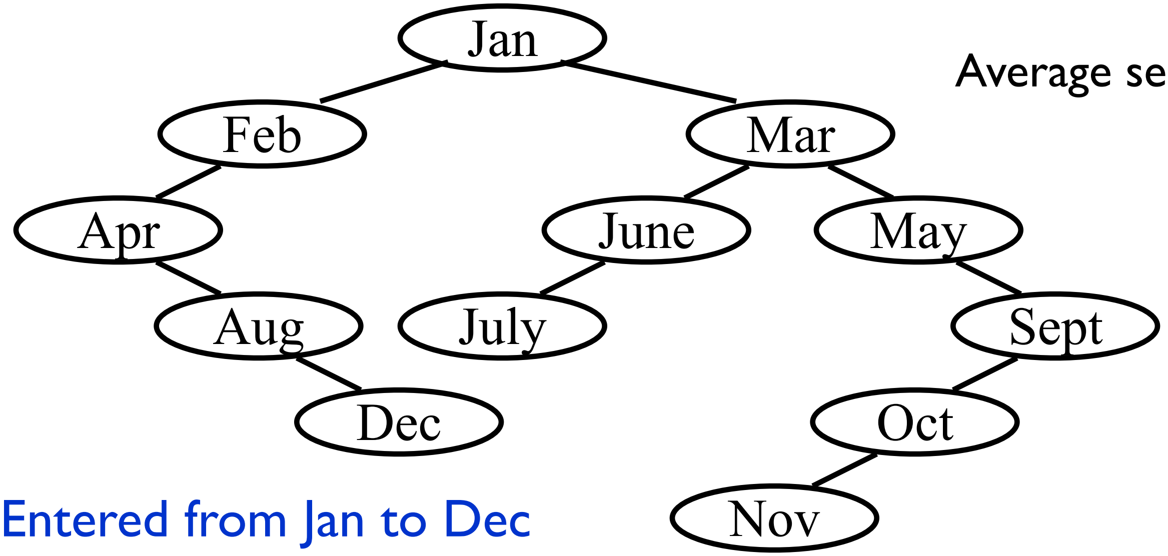
Entered from Jan to Dec



Average search time = 3.1

A balanced tree

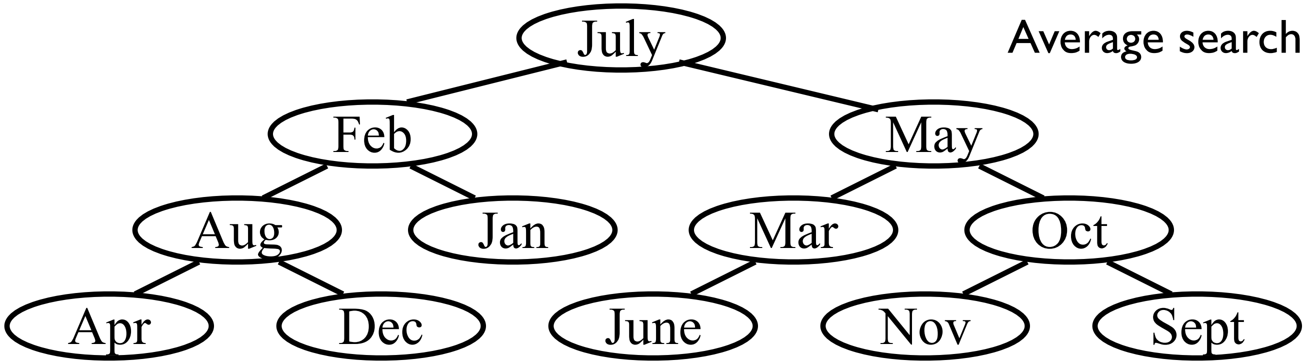
〚Example〛 2 binary search trees obtained for the months of the year



Average search time = 3.5

Average search time of the skew tree = 6.5

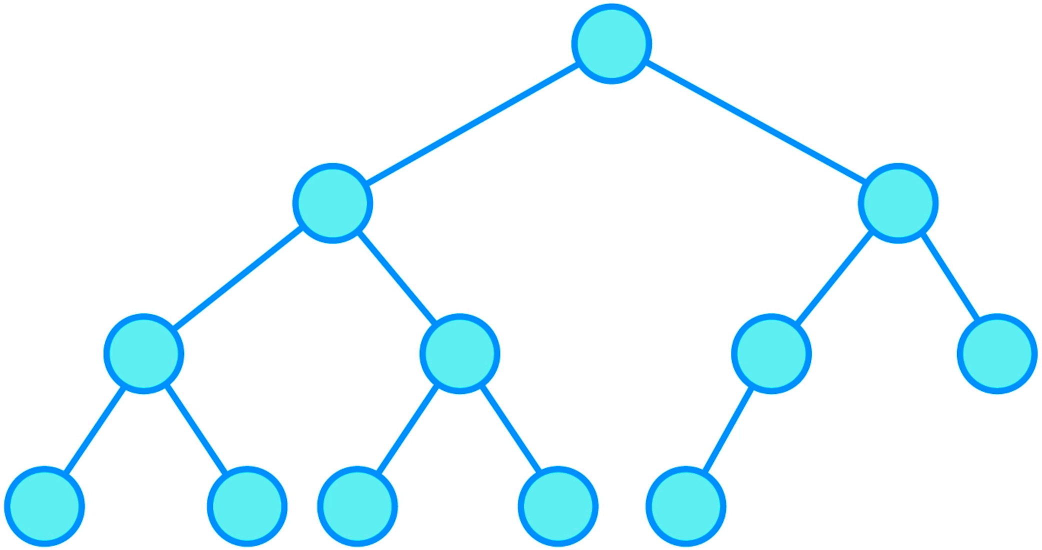
Entered from Jan to Dec



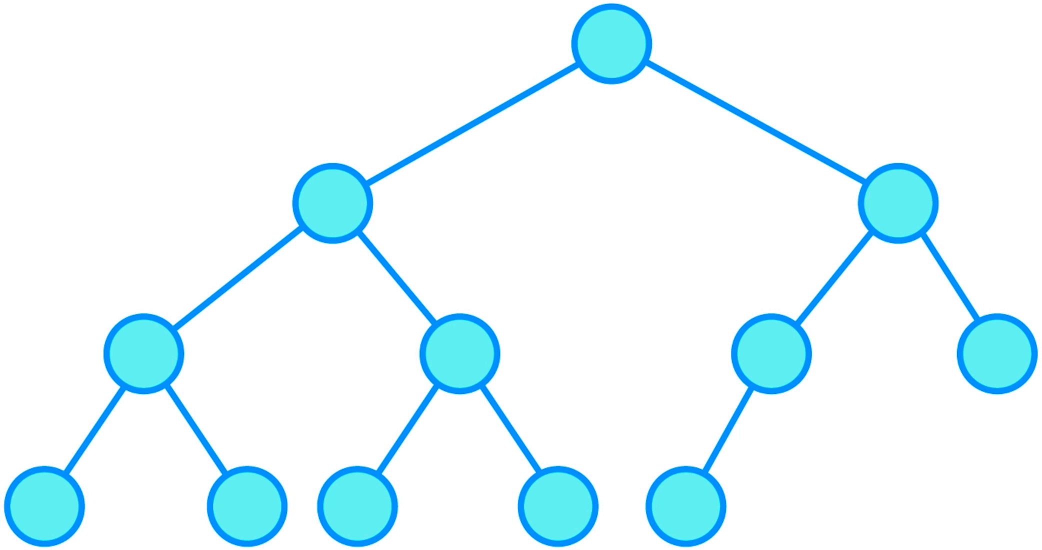
Average search time = 3.1

A balanced tree

Why Not Use Complete BST?



Why Not Use Complete BST?

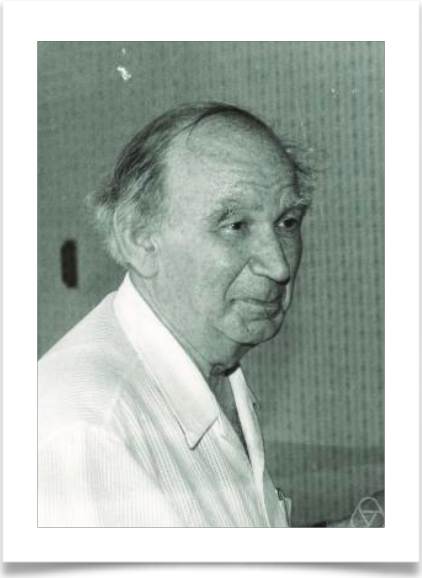


The constraint is too strong.
BST needs to preserve instance order,
every operation involves global tuning of the structure.
We should relax the constraint.

Outline: Balanced Binary Search Trees (1)

- Binary search trees
- AVL trees
- Splay trees
- Amortized analysis
- Take-home messages

Adelson-Velskii-Landis (AVL) Trees (1962)



- Self-balanced trees which dynamically modifies tree structure to keep the tree balanced during operations.

Figure courtesy: https://www.chessprogramming.org/Georgy_Adelson-Velsky
https://en.wikipedia.org/wiki/Evgenii_Landis

Adelson-Velskii-Landis (AVL) Trees (1962)

Figure courtesy: https://www.chessprogramming.org/Georgy_Adelson-Velsky

AVL Trees

【Definition】An empty binary tree is height-balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is height-balanced iff

(1) T_L and T_R are height balanced, and

(2) $| h_L - h_R | \leq 1$ where h_L and h_R are the heights of T_L and T_R , respectively.

AVL Trees

The height of an empty tree is defined to be –1.

【Definition】 An empty binary tree is height-balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is height-balanced iff

(1) T_L and T_R are height balanced, and

(2) $| h_L - h_R | \leq 1$ where h_L and h_R are the heights of T_L and T_R , respectively.

AVL Trees

The height of an empty tree is defined to be –1.

【Definition】An empty binary tree is height-balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is height-balanced iff

(1) T_L and T_R are height balanced, and

(2) $| h_L - h_R | \leq 1$ where h_L and h_R are the heights of T_L and T_R , respectively.

【Definition, AVL tree】The balance factor $BF(\text{ node }) = h_L - h_R$. In an AVL tree, $BF(\text{ node }) = -1, 0,$ or 1.

AVL Trees

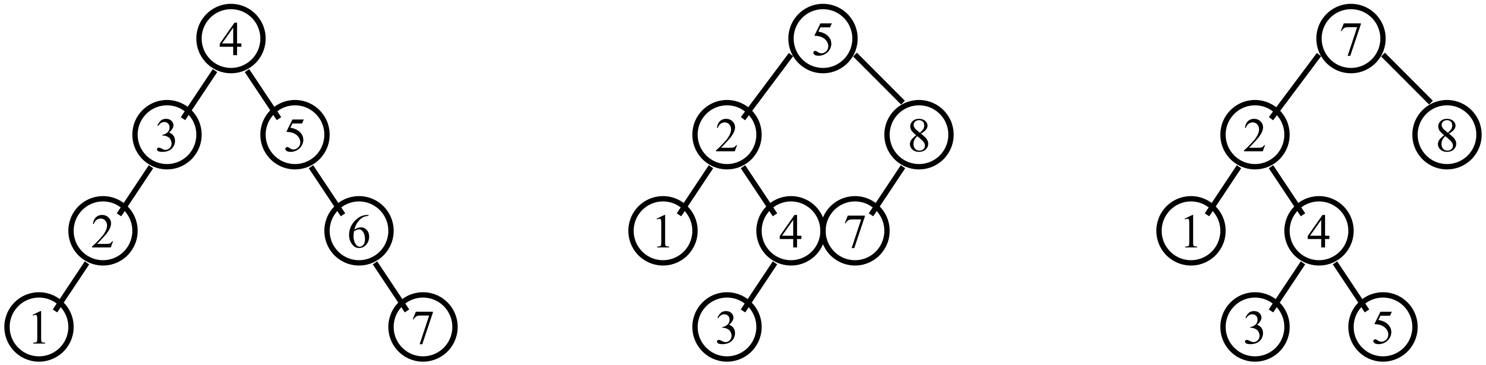
The height of an empty tree is defined to be –1.

【Definition】An empty binary tree is height-balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is height-balanced iff

(1) T_L and T_R are height balanced, and

(2) $| h_L - h_R | \leq 1$ where h_L and h_R are the heights of T_L and T_R, respectively.

【Definition, AVL tree】The balance factor $BF(\text{ node }) = h_L - h_R$. In an AVL tree, $BF(\text{ node }) = -1, 0,$ or 1.



AVL Trees

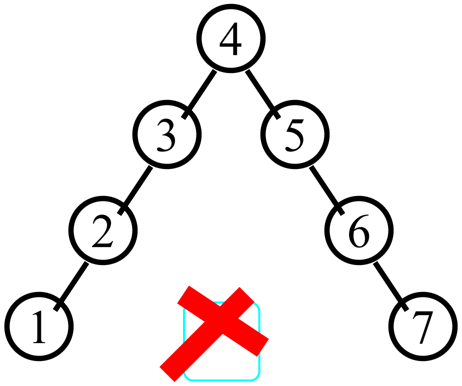
The height of an empty tree is defined to be –1.

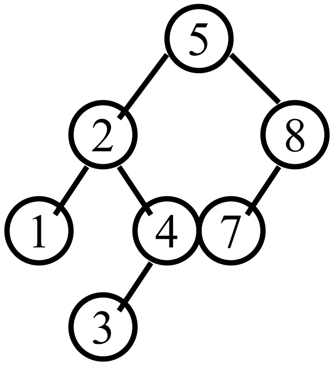
【Definition】An empty binary tree is height-balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is height-balanced iff

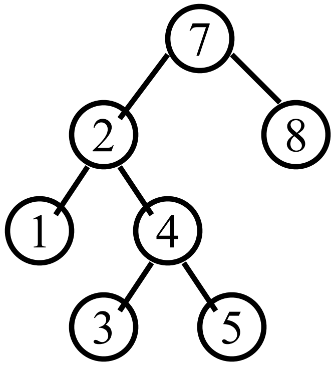
(1) T_L and T_R are height balanced, and

(2) $| h_L - h_R | \leq 1$ where h_L and h_R are the heights of T_L and T_R , respectively.

【Definition, AVL tree】The balance factor BF(node) = $h_L - h_R$. In an AVL tree, BF(node) = −1, 0, or 1.







AVL Trees

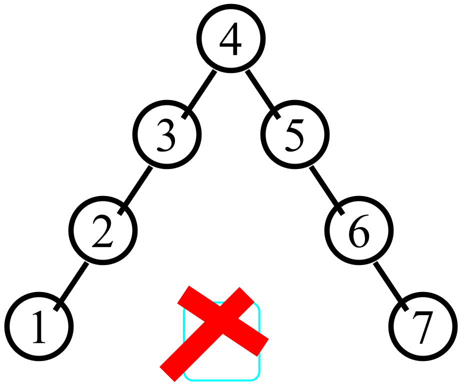
The height of an empty tree is defined to be -1.

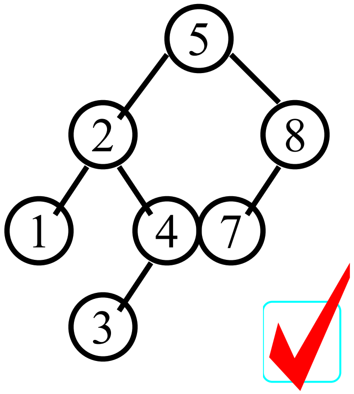
【Definition】 An empty binary tree is height-balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is height-balanced iff

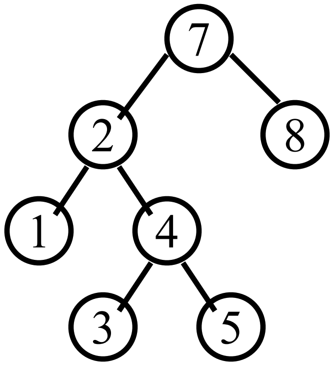
(1) T_L and T_R are height balanced, and

(2) $| h_L - h_R | \leq 1$ where h_L and h_R are the heights of T_L and T_R, respectively.

【Definition, AVL tree】 The balance factor $BF(\text{ node }) = h_L - h_R$. In an AVL tree, $BF(\text{ node }) = -1, 0,$ or 1.







AVL Trees

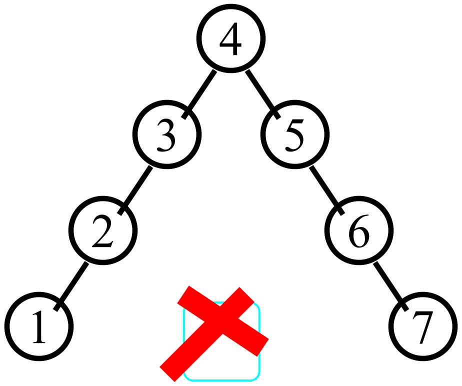
The height of an empty tree is defined to be –1.

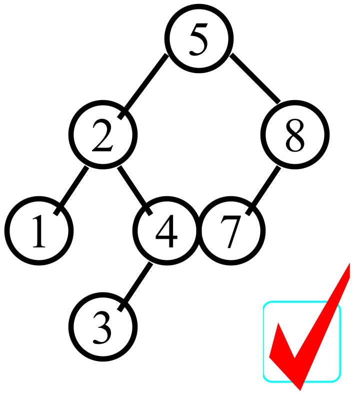
【Definition】 An empty binary tree is height-balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is height-balanced iff

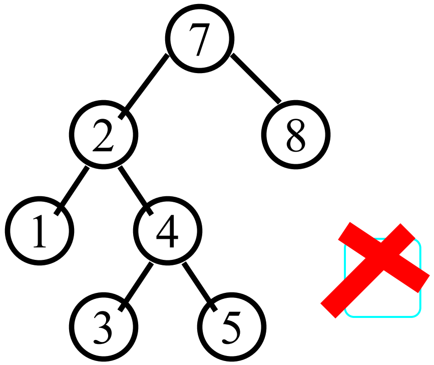
(1) T_L and T_R are height balanced, and

(2) $| h_L - h_R | \leq 1$ where h_L and h_R are the heights of T_L and T_R , respectively.

【Definition, AVL tree】 The balance factor $BF(\text{ node }) = h_L - h_R$. In an AVL tree, $BF(\text{ node }) = -1, 0,$ or 1.

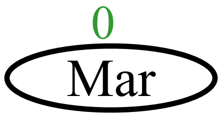




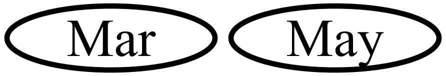


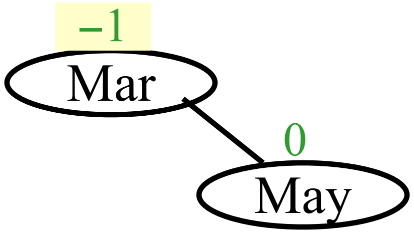
〖Example〗 Input the months

〖Example〗 Input the months

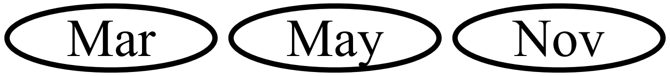


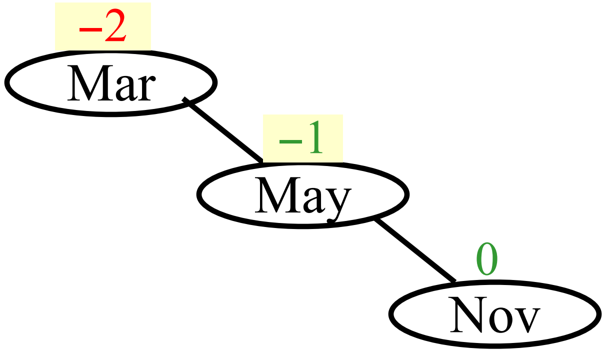
〖Example〗 Input the months



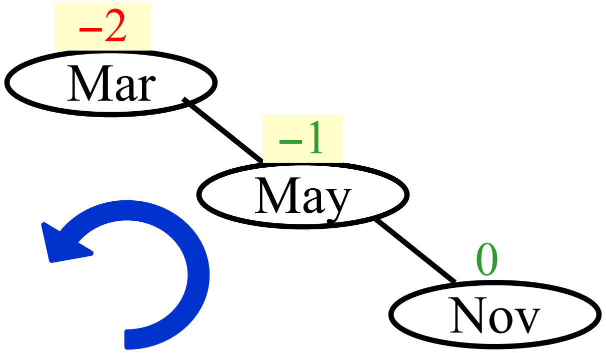


〖Example〗 Input the months

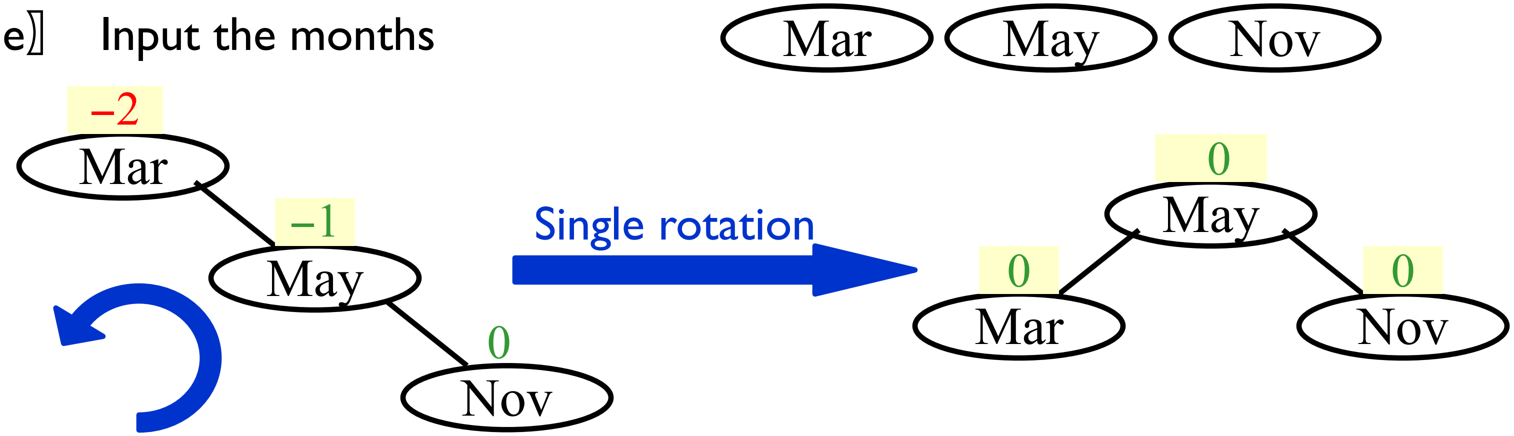




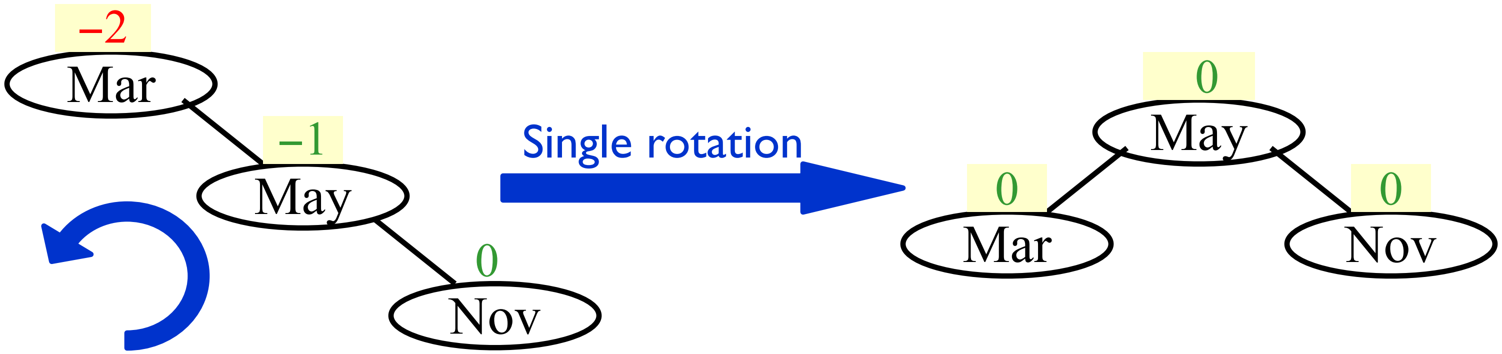
〖Example〗 Input the months



〚Example〛 Input the months

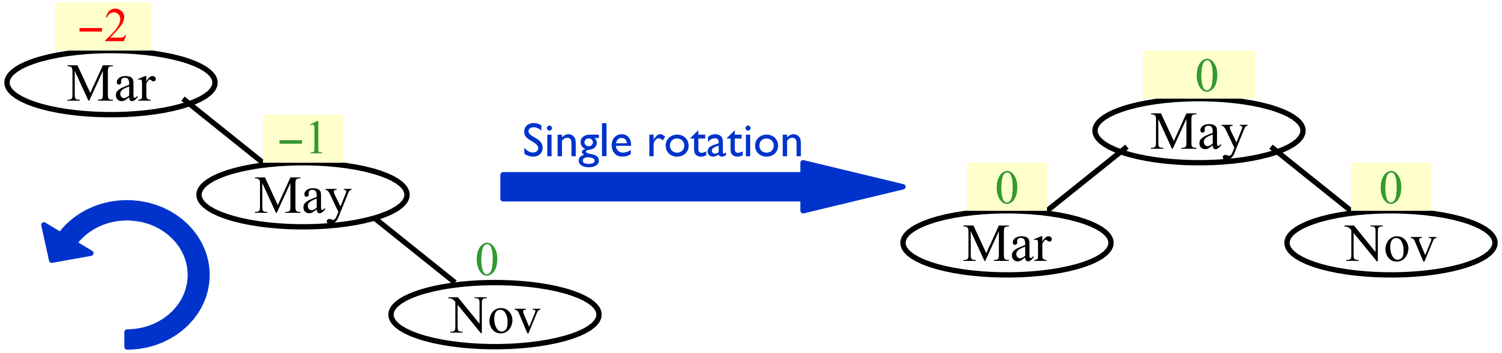


〚Example〛 Input the months



👁 The trouble maker Nov is in the right subtree's right subtree of the trouble finder Mar. Hence it is called an RR rotation.

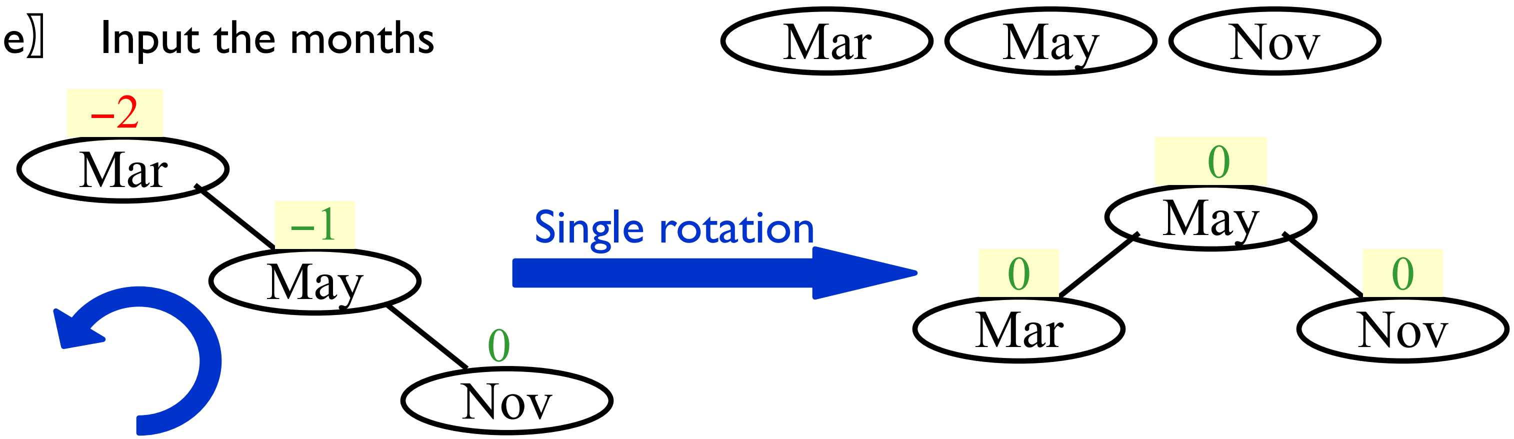
〚Example〛 Input the months



The trouble maker Nov is in the right subtree's right subtree of the trouble finder Mar. Hence it is called an RR rotation.

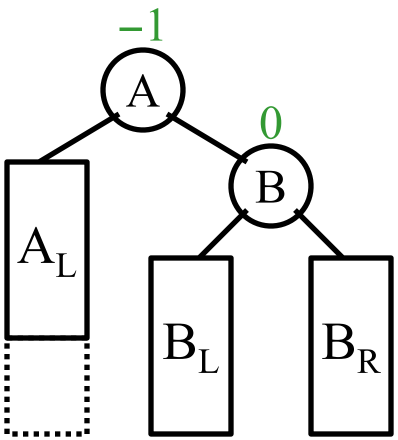
In general:

〚Example〛 Input the months

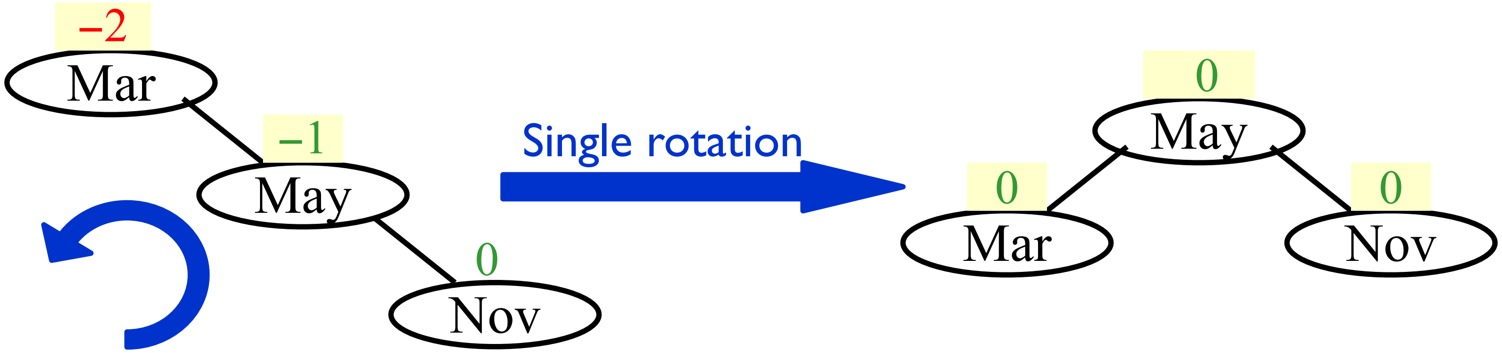


The trouble maker Nov is in the right subtree's right subtree of the trouble finder Mar. Hence it is called an RR rotation.

In general:

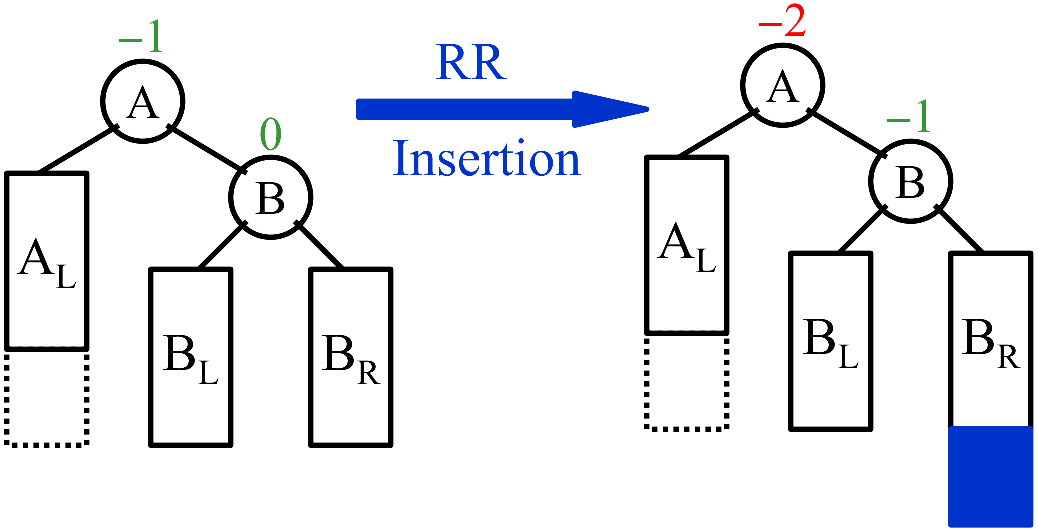


〖Example〗 Input the months Mar May Nov

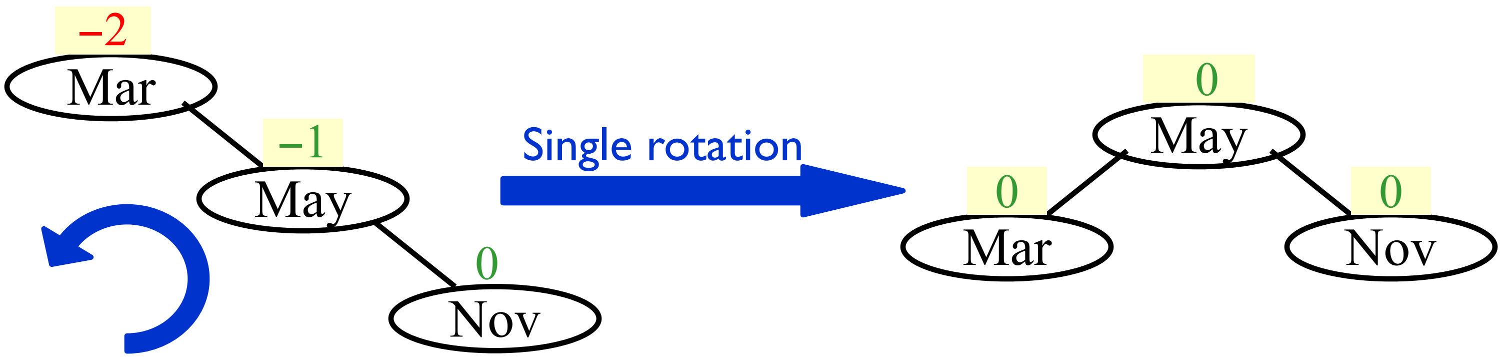


The trouble maker Nov is in the right subtree's right subtree of the trouble finder Mar. Hence it is called an RR rotation.

In general:

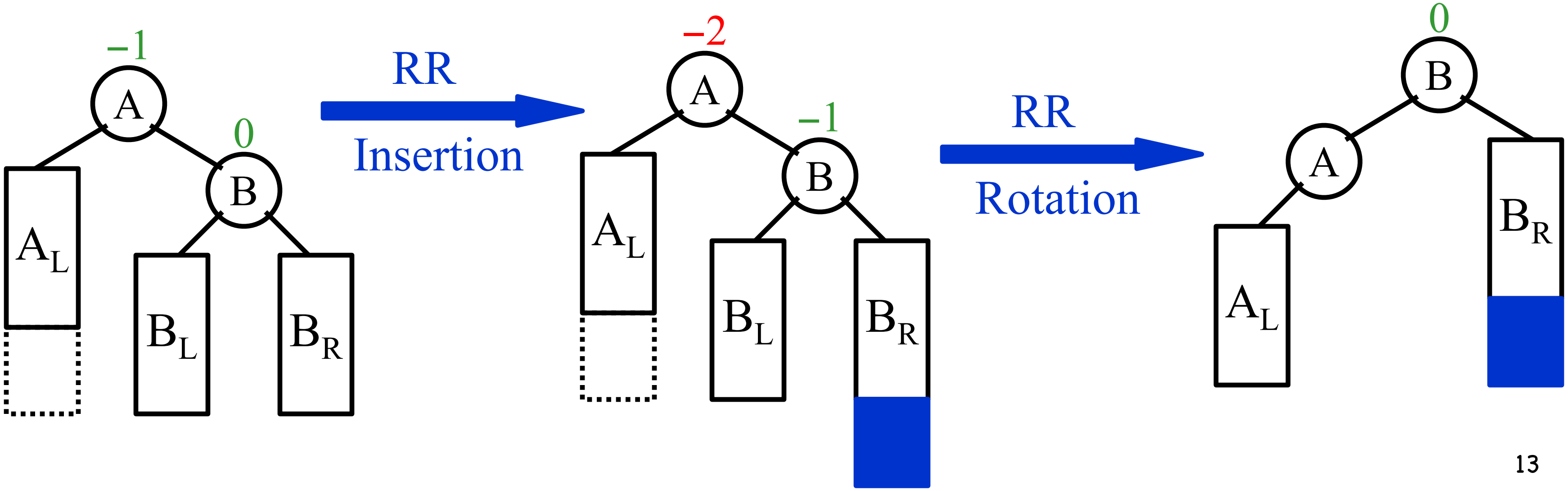


〖Example〗 Input the months Mar May Nov

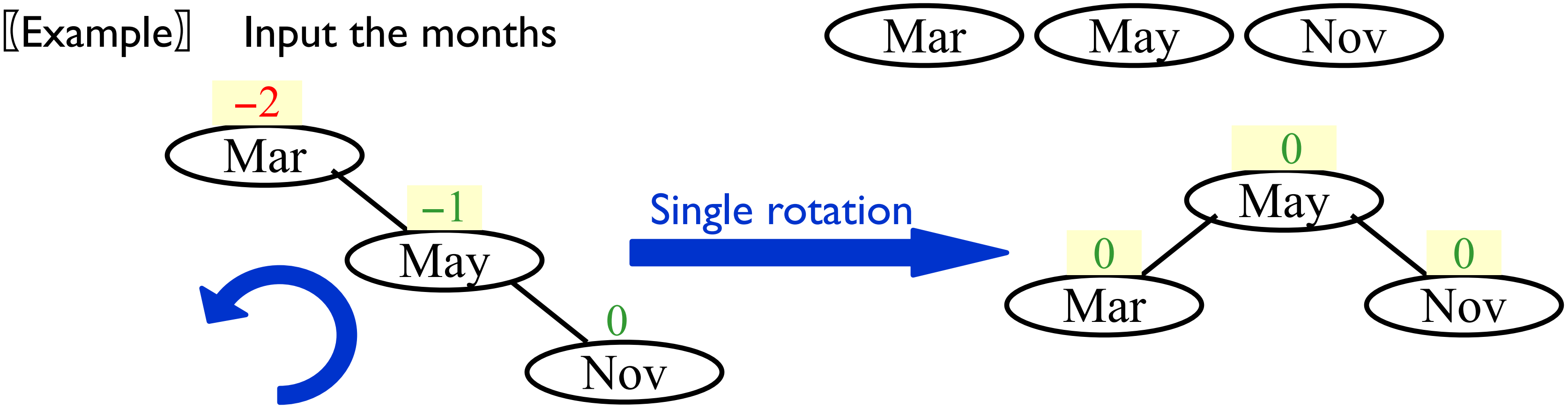


The trouble maker Nov is in the right subtree's right subtree of the trouble finder Mar. Hence it is called an RR rotation.

In general:

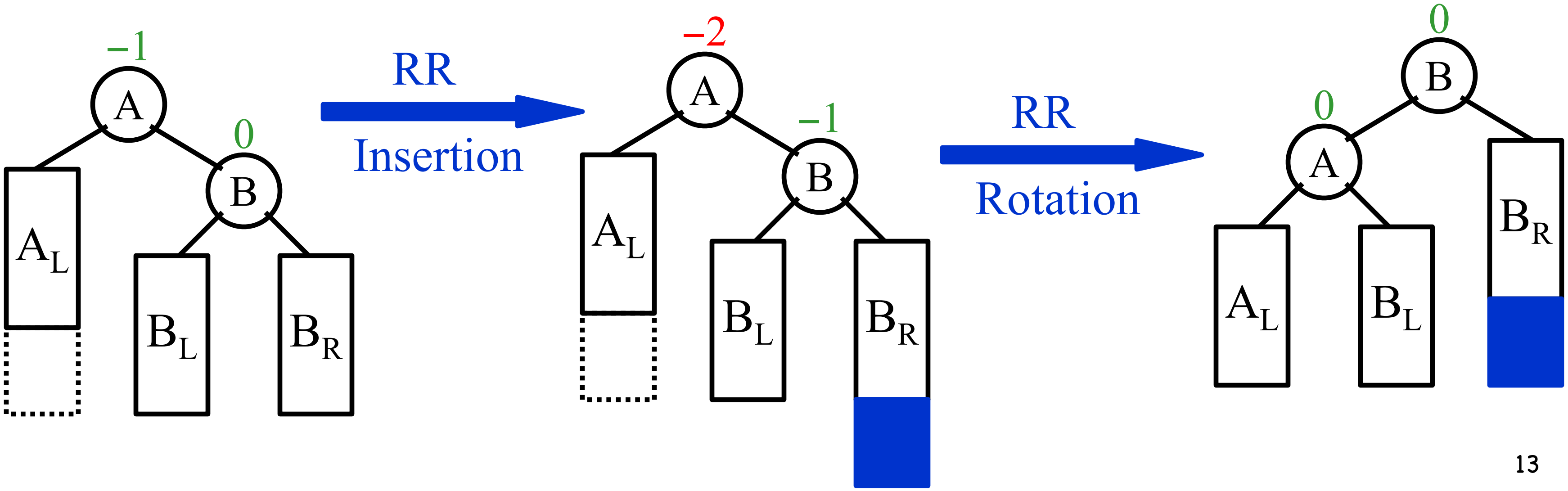


〖Example〗 Input the months

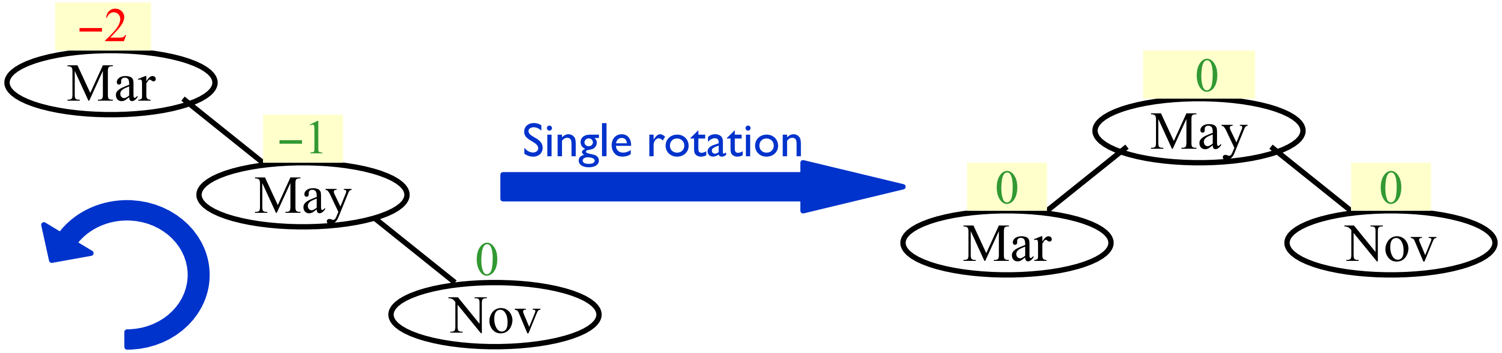


The trouble maker Nov is in the right subtree's right subtree of the trouble finder Mar. Hence it is called an RR rotation.

In general:

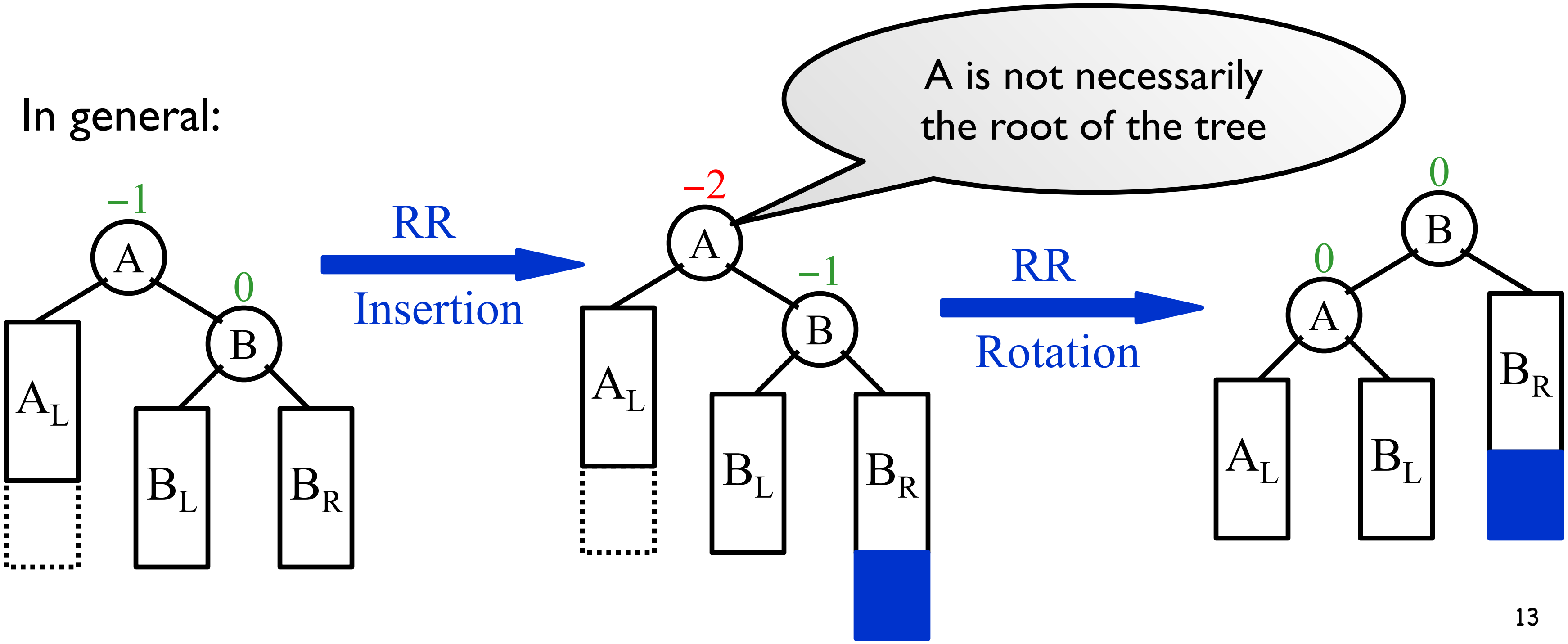


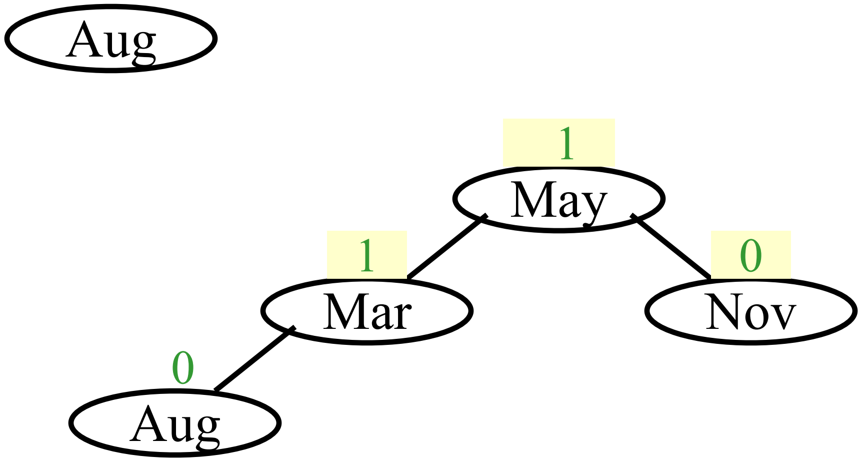
〚Example〛 Input the months

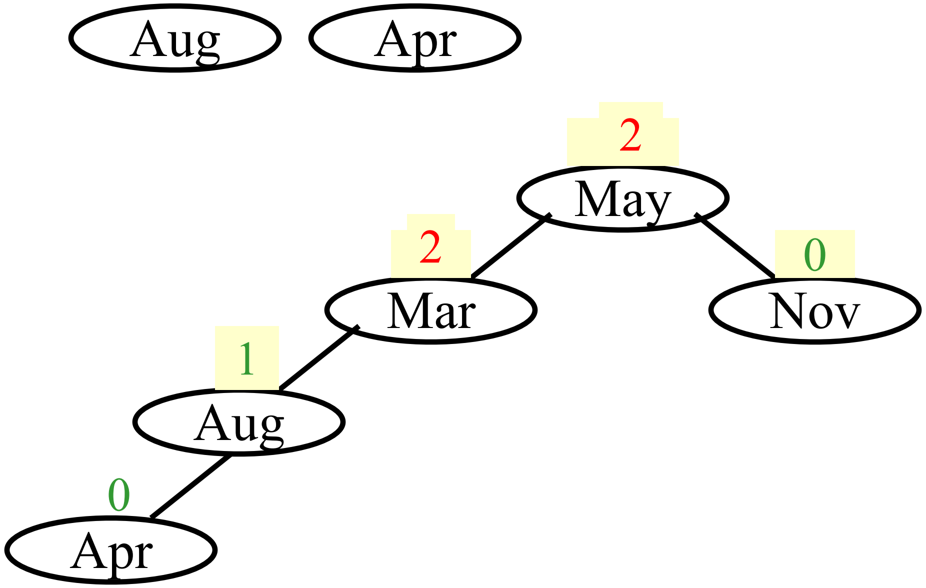

Single rotation

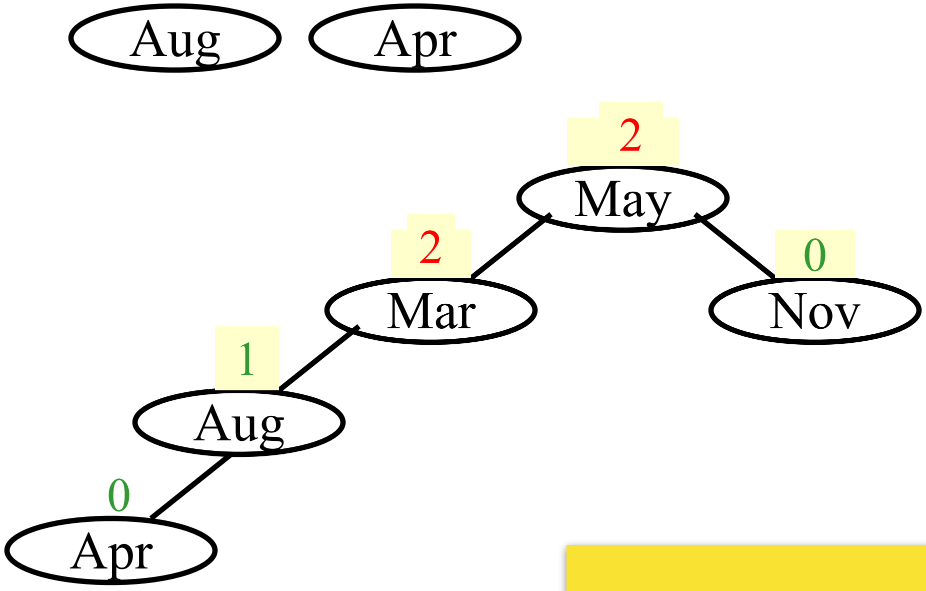
The trouble maker Nov is in the right subtree's right subtree of the trouble finder Mar. Hence it is called an RR rotation.

In general:

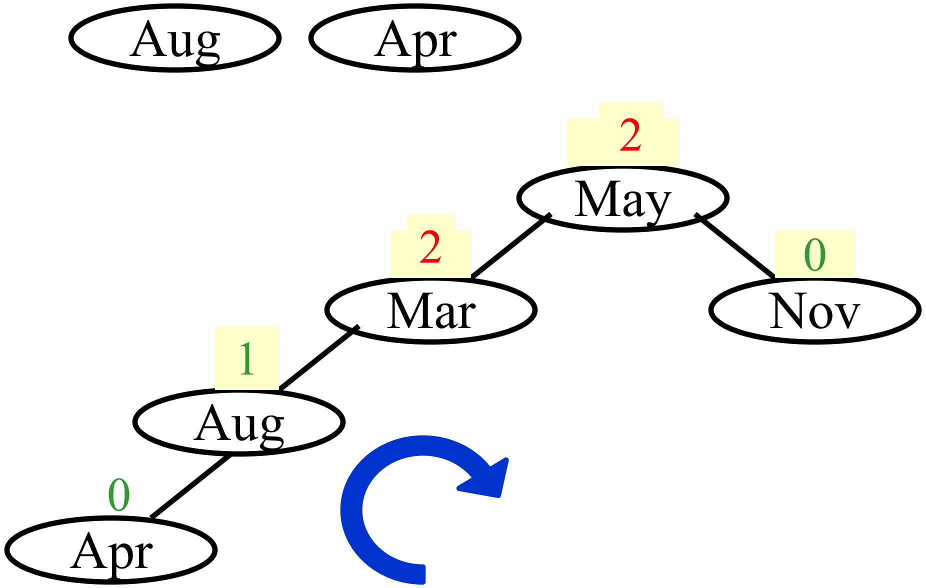

A is not necessarily the root of the tree

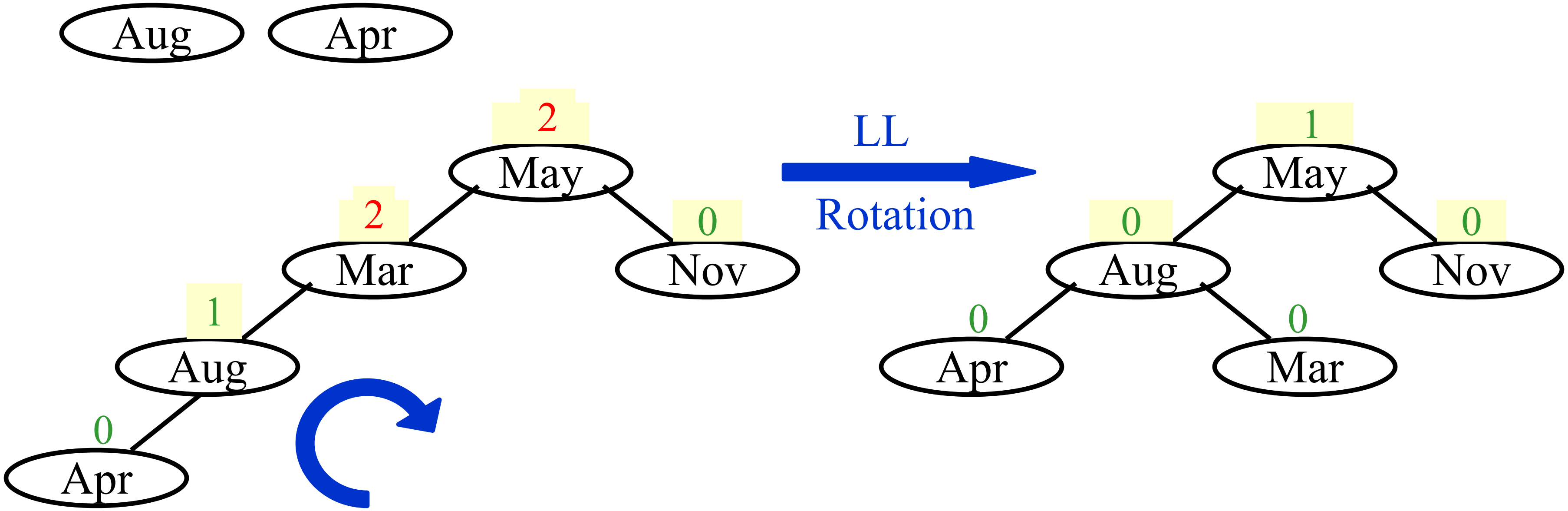

Aug
1
May
1
Mar
0
Nov
0
Aug

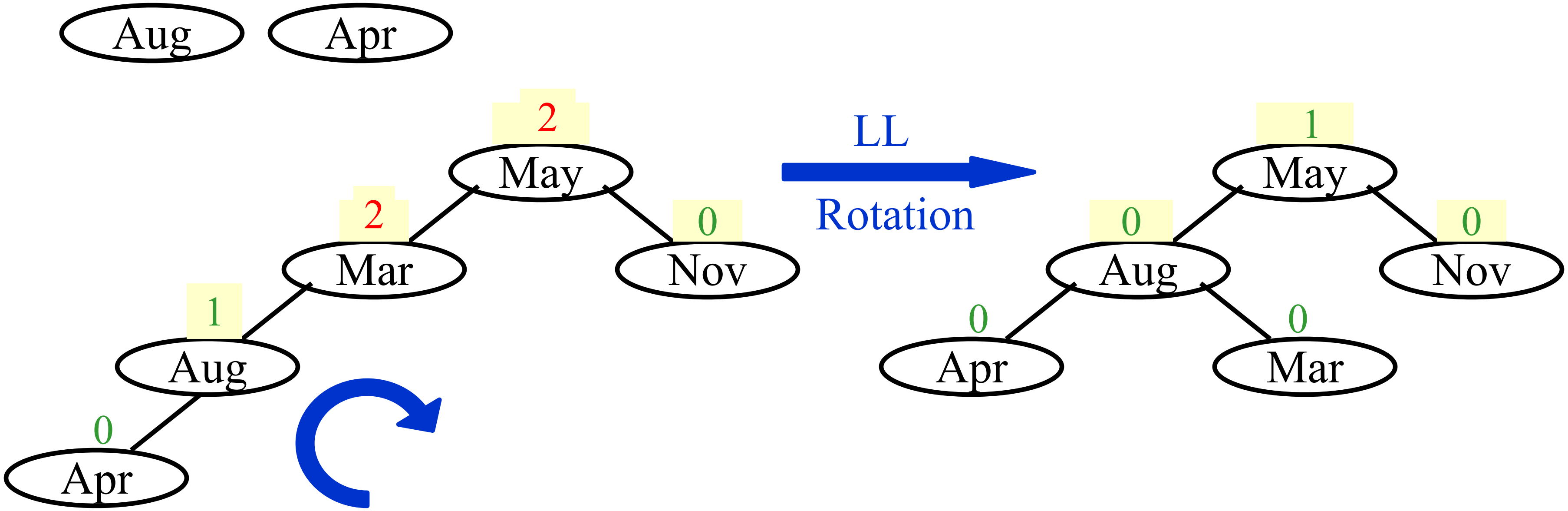

Aug
Apr
2
May
2
Mar
0
Nov
1
Aug
0
Apr



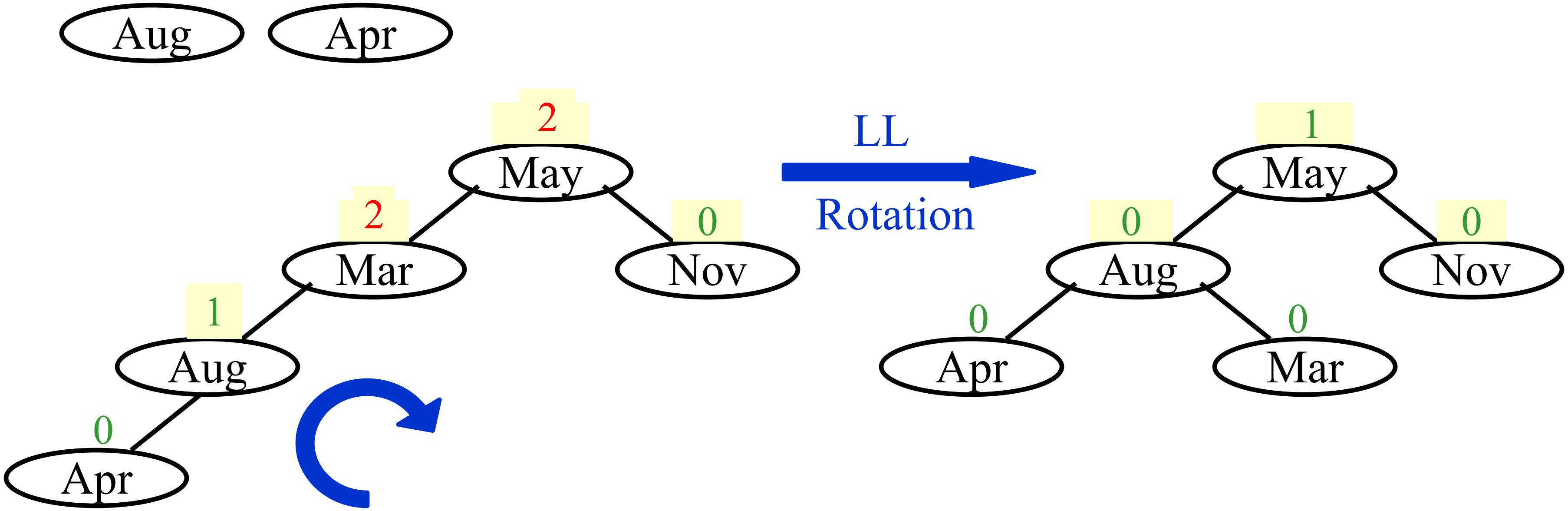
What can we do now?


Aug
Apr
2
May
2
Mar
0
Nov
1
Aug
0
Apr


Aug
Apr
2
May
2
Mar
0
Nov
1
Aug
0
Apr
LL
Rotation
1
May
0
Aug
0
Nov
0
Apr
0
Mar


Aug
Apr
2
May
2
Mar
0
Nov
1
Aug
0
Apr
LL
Rotation
1
May
0
Aug
0
Nov
0
Apr
0
Mar

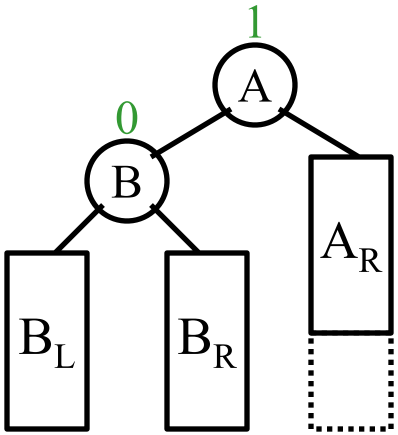
In general:

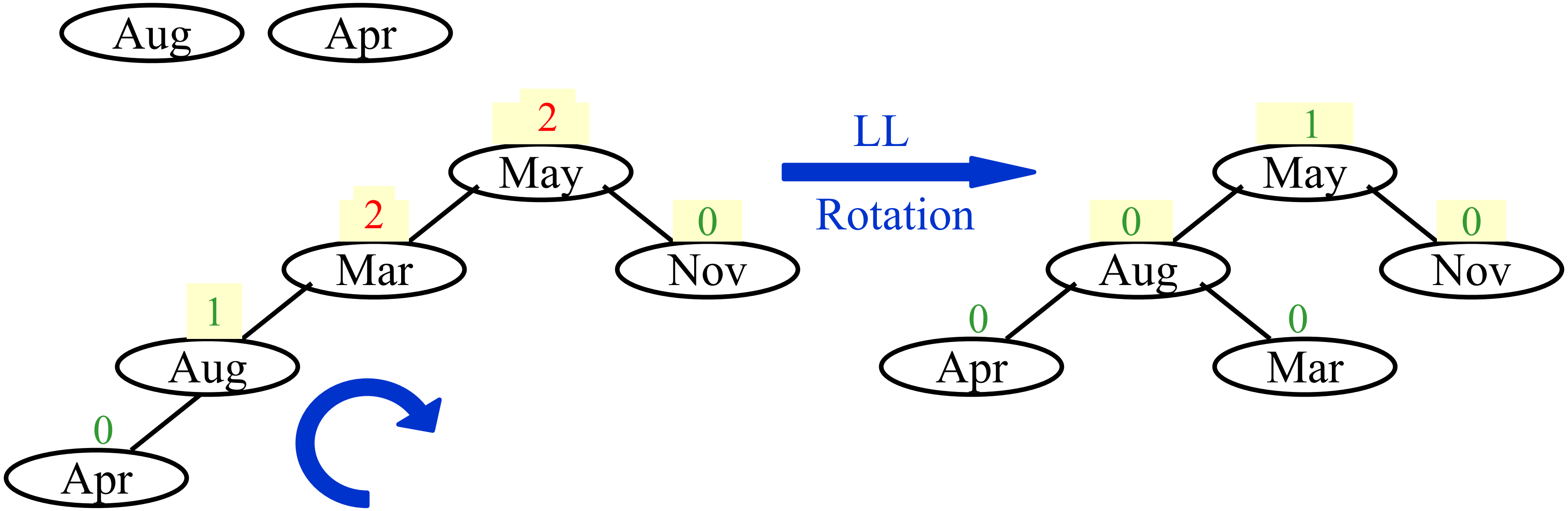


LL

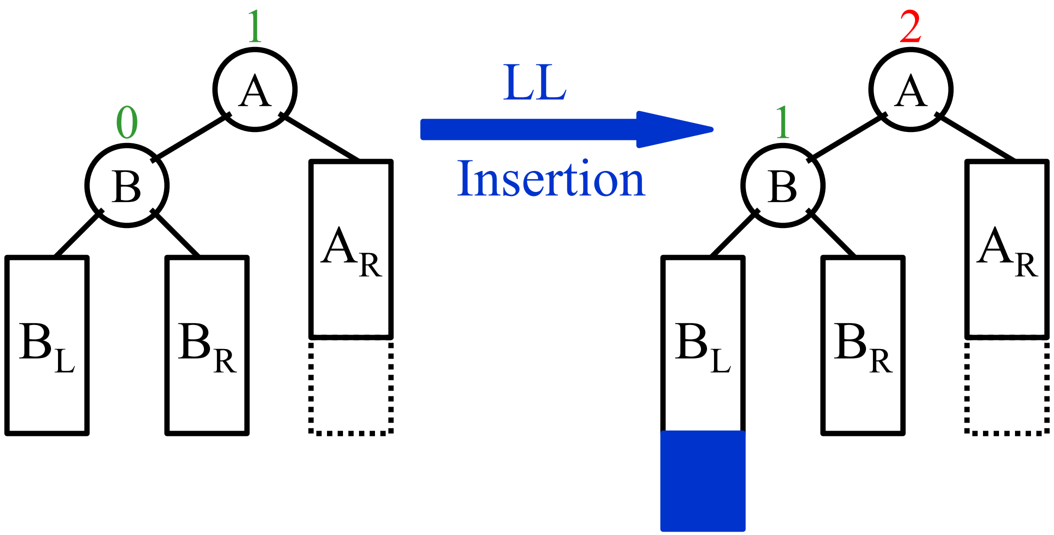
Rotation

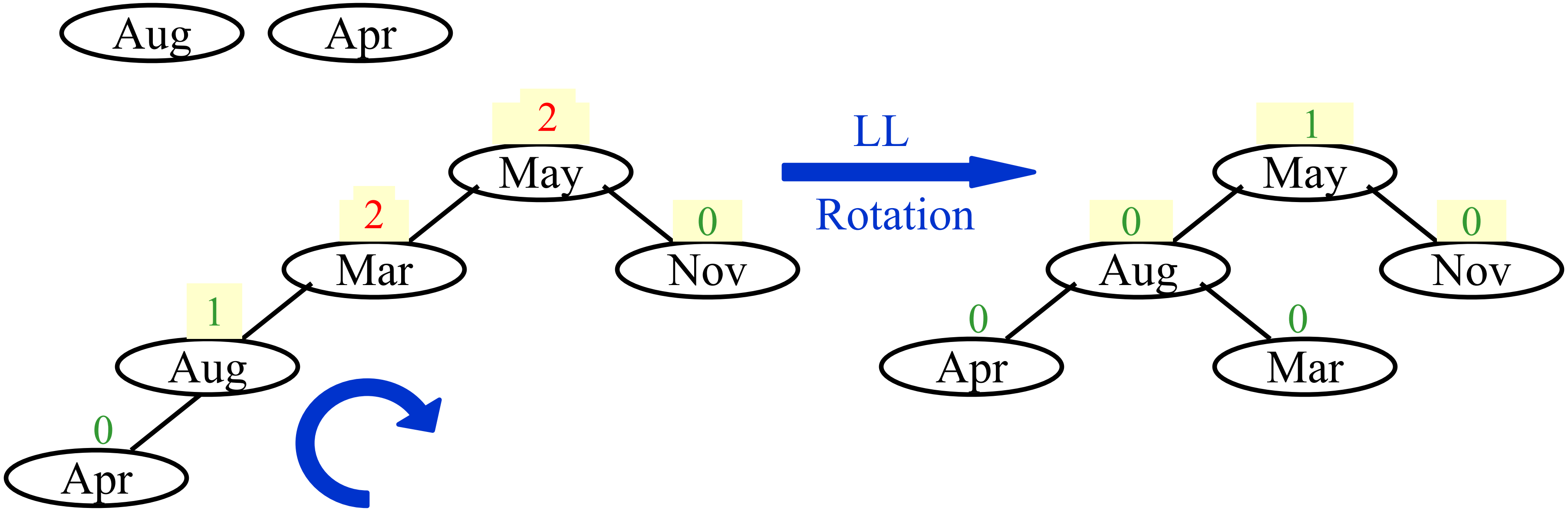
In general:



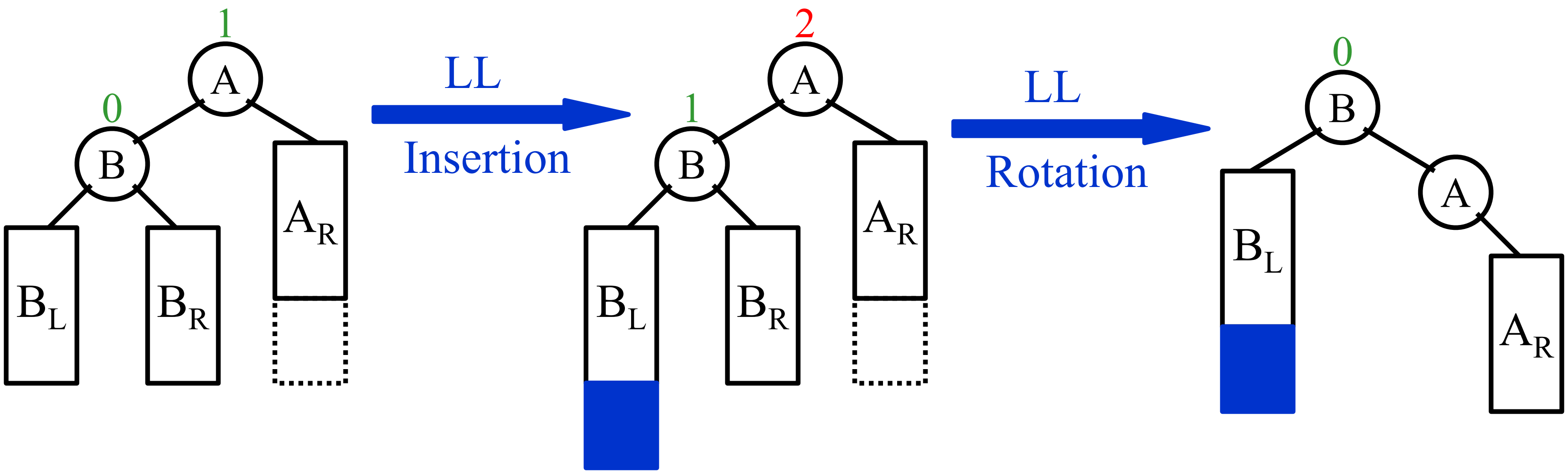

Aug
Apr
2
May
2
Mar
0
Nov
1
Aug
0
Apr
LL
Rotation
1
May
0
Aug
0
Nov
0
Apr
0
Mar

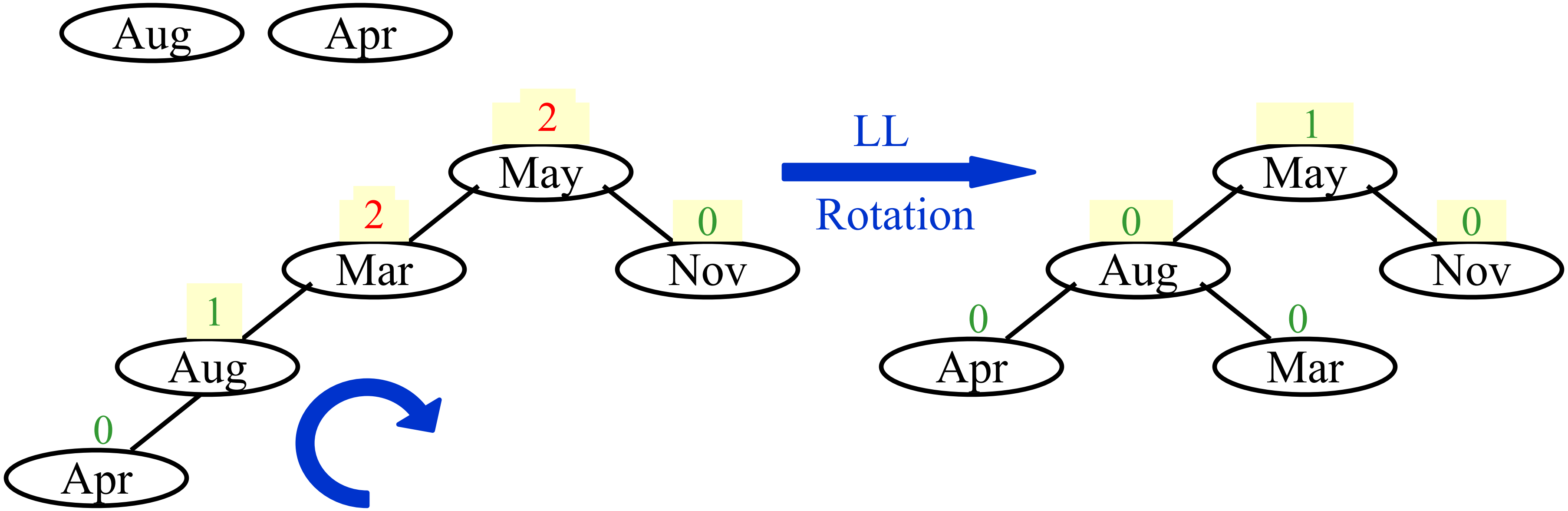
In general:


1
A
0
B
B_L
B_R
A_R
LL
Insertion
2
A
1
B
B_L
B_R
A_R

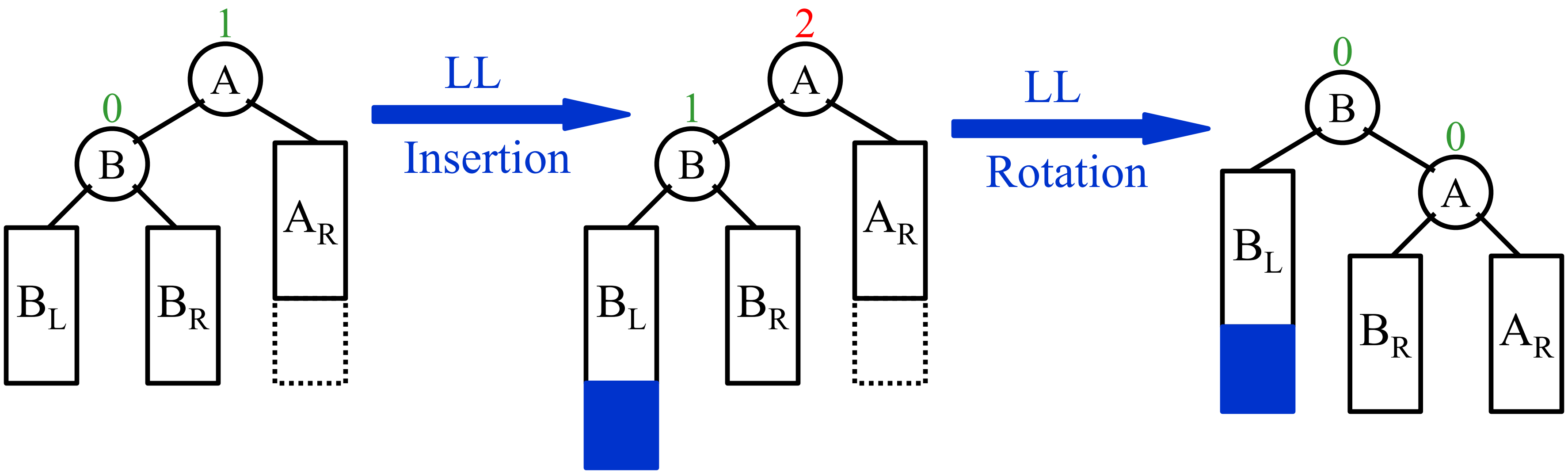

Aug
Apr
2
May
2
Mar
0
Nov
1
Aug
0
Apr
LL
Rotation
1
May
0
Aug
0
Nov
0
Apr
0
Mar

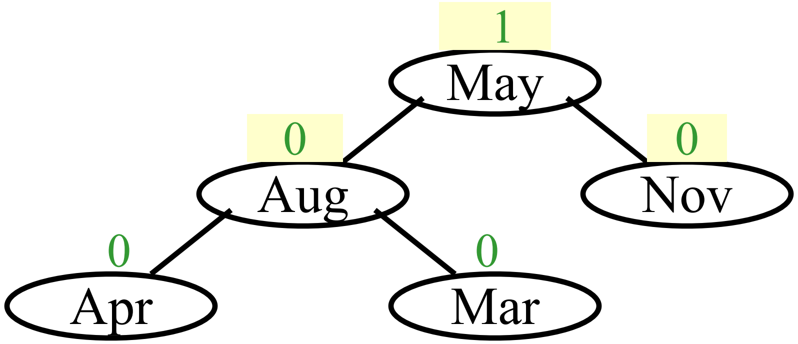
In general:

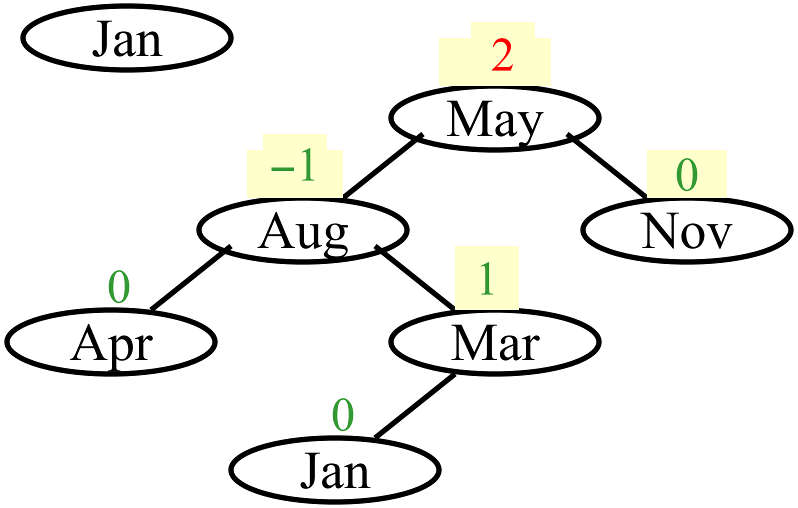

1
A
0
B
B_L
B_R
A_R
LL
Insertion
2
A
1
B
B_L
B_R
A_R
LL
Rotation
0
B
A
B_L
A_R

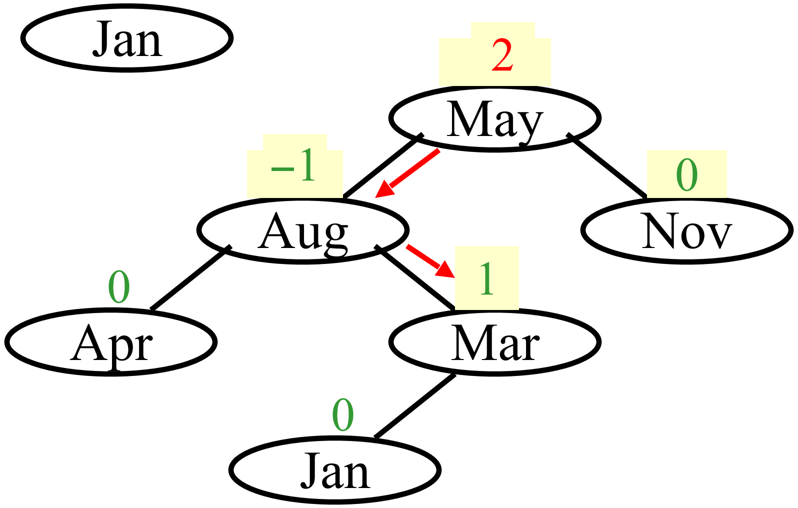

Aug
Apr
2
May
2
Mar
0
Nov
1
Aug
0
Apr
LL
Rotation
1
May
0
Aug
0
Nov
0
Apr
0
Mar

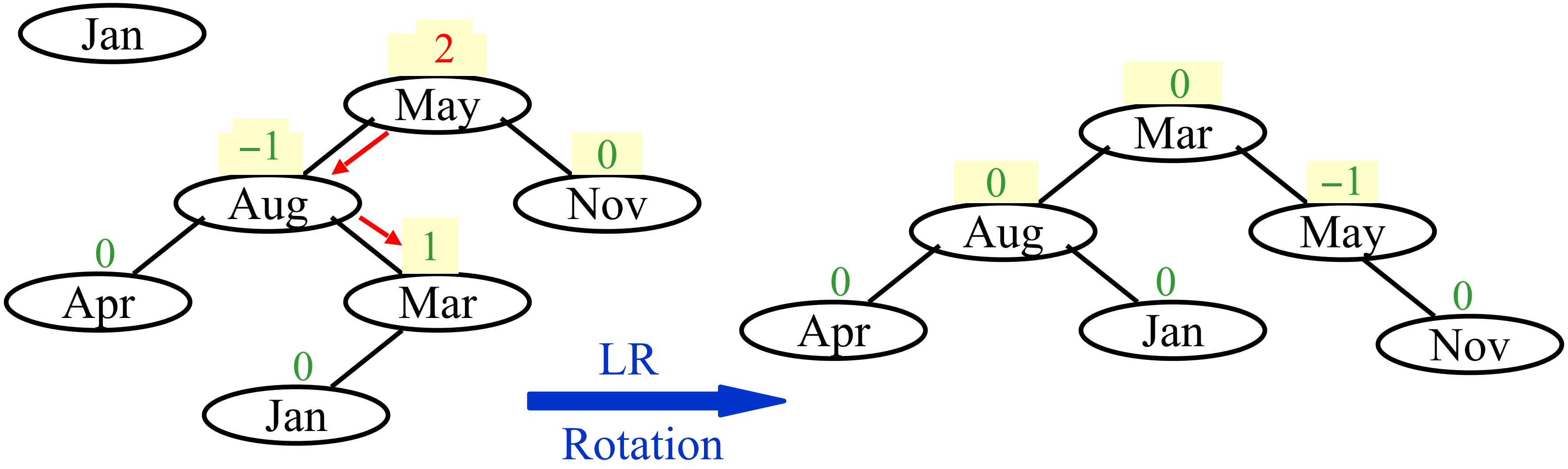
In general:

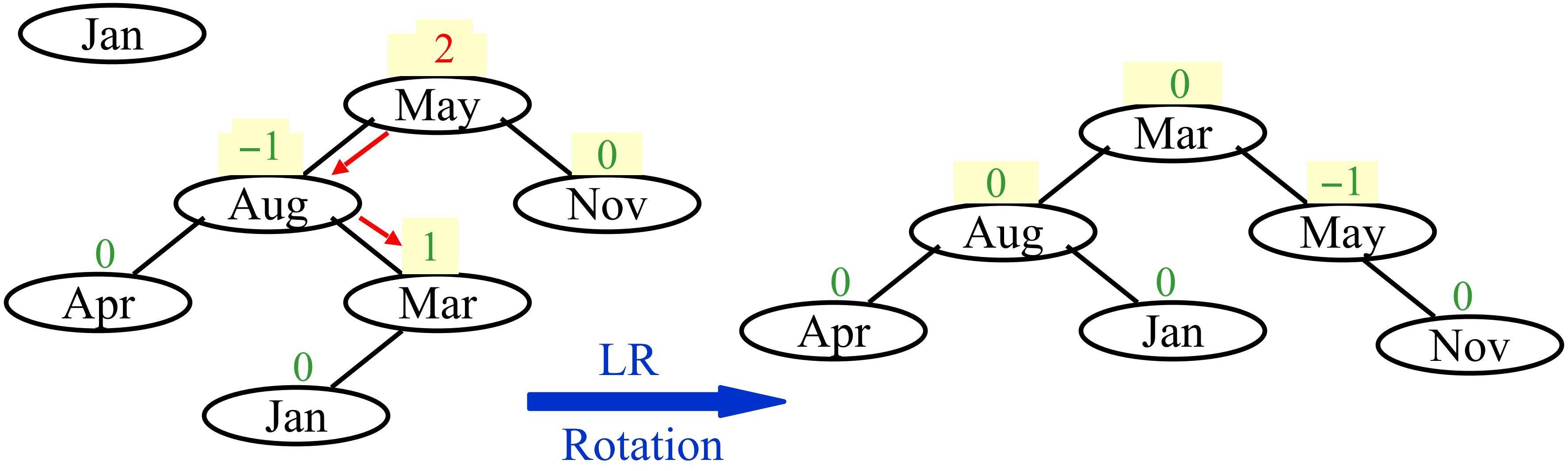

1
A
0
B
B_L
B_R
A_R
LL
Insertion
2
A
1
B
B_L
B_R
A_R
LL
Rotation
0
B
0
A
B_L
B_R
A_R


1
May
0
Aug
0
Nov
0
Apr
0
Mar

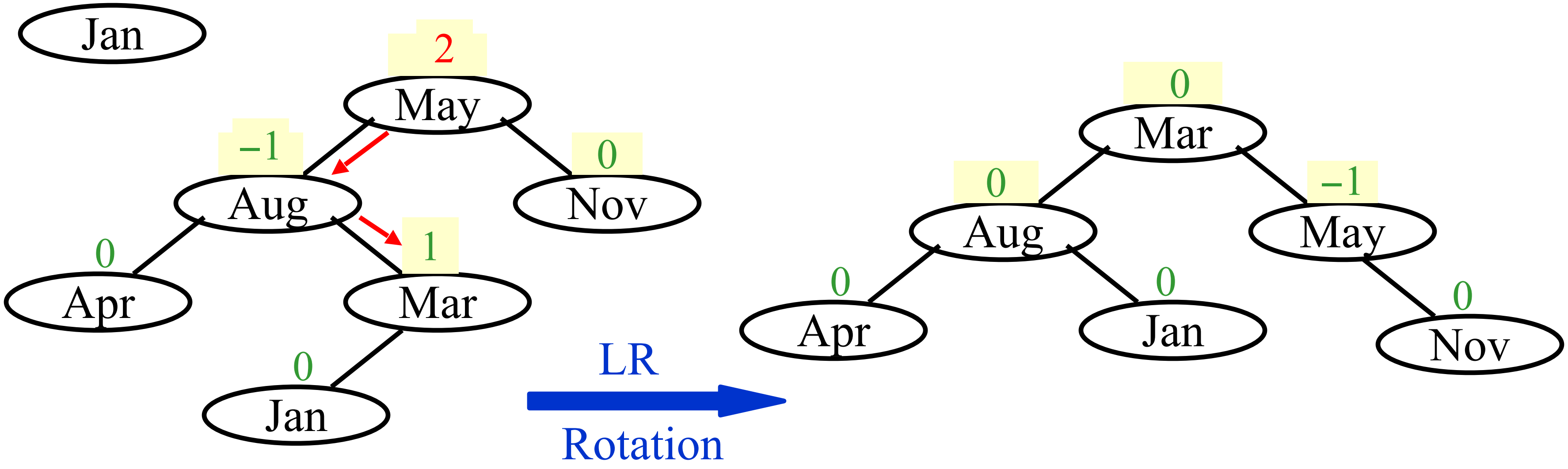

Jan
2
May
-1
Aug
0
Nov
0
Apr
1
Mar
0
Jan


Jan
2
May
−1
Aug
0
Nov
0
Apr
1
Mar
0
Jan

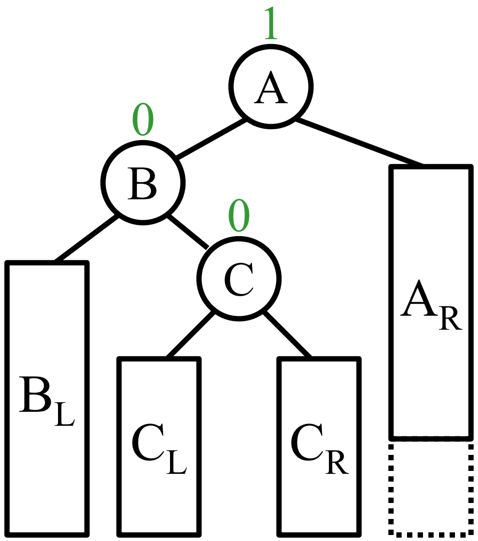

Jan
2
May
–1
Aug
0
Nov
0
Apr
1
Mar
0
Jan
LR
Rotation
0
Mar
0
Aug
–1
May
0
Apr
0
Jan
0
Nov

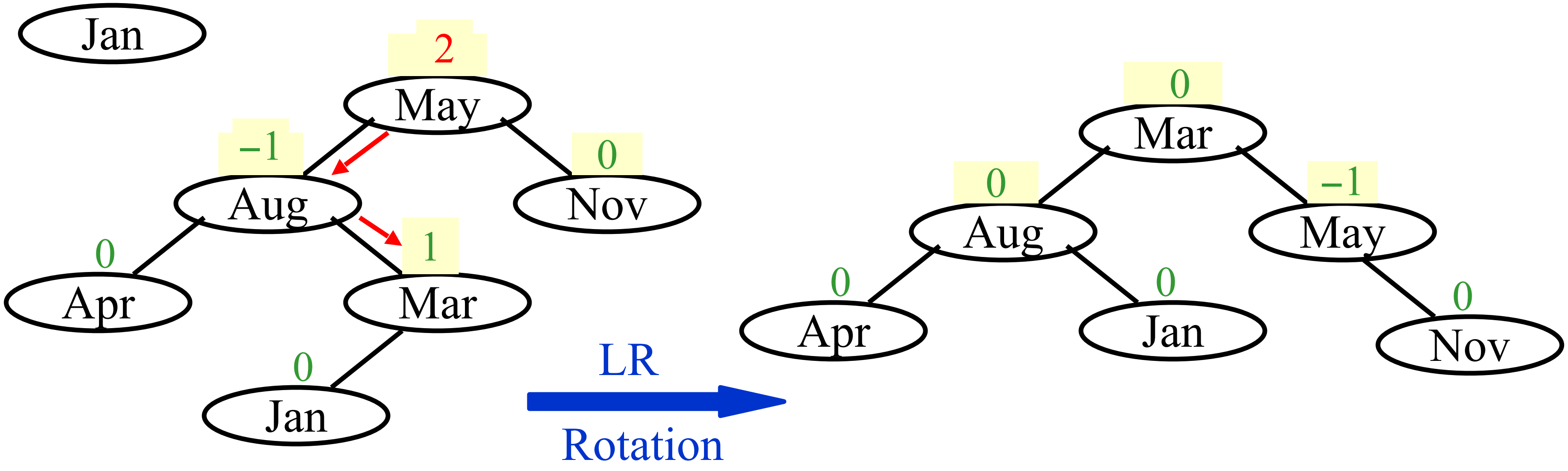

Jan
2
May
−1
Aug
0
Nov
0
Apr
1
Mar
0
Jan
LR
Rotation
0
Mar
0
Aug
−1
May
0
Apr
0
Jan
0
Nov

In general:

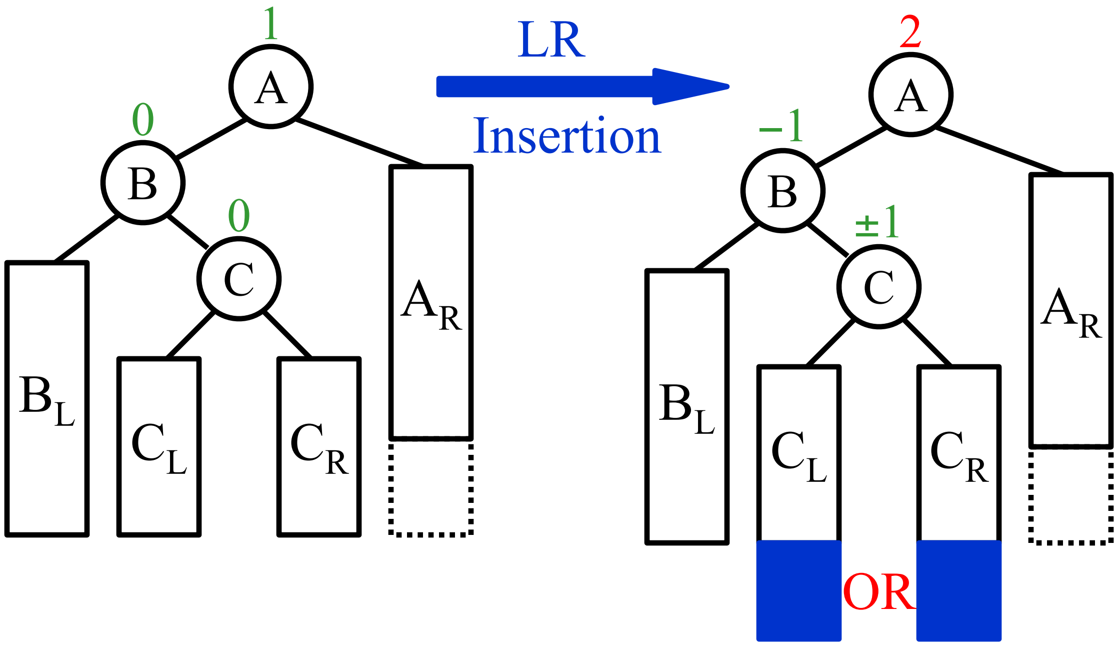

Jan
2
May
–1
Aug
0
Nov
0
Apr
1
Mar
0
Jan
LR
Rotation
0
Mar
0
Aug
–1
May
0
Apr
0
Jan
0
Nov

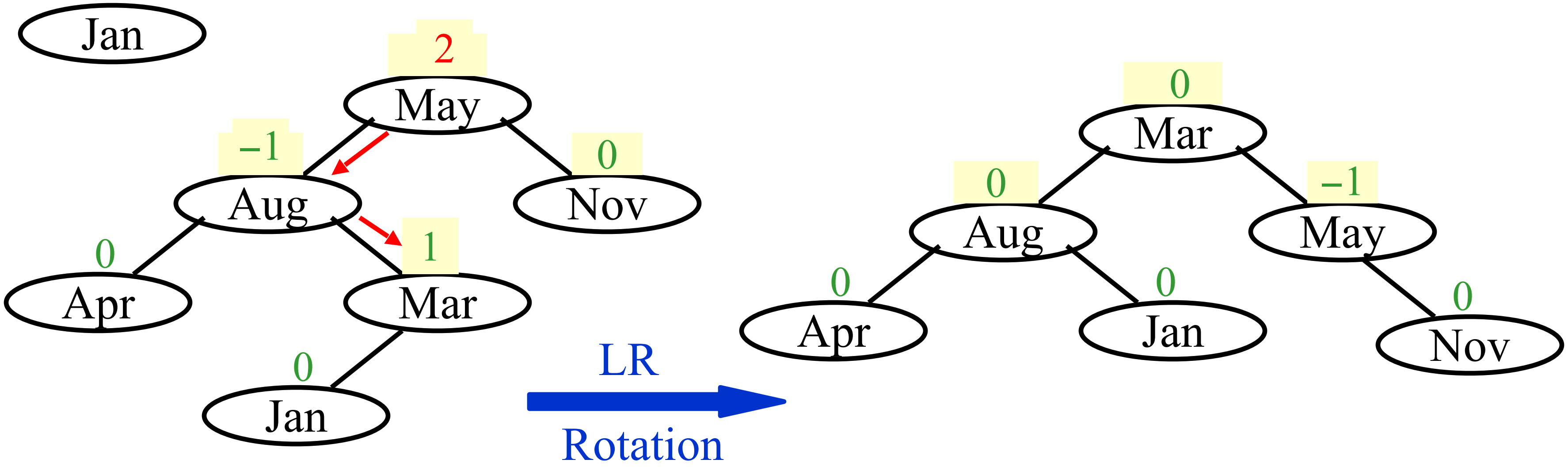
In general:


1
A
0
B
0
C
B_L
C_L
C_R
A_R


Jan
2
May
−1
Aug
0
Nov
0
Apr
1
Mar
0
Jan
LR
Rotation
0
Mar
0
Aug
−1
May
0
Apr
0
Jan
0
Nov

In general:

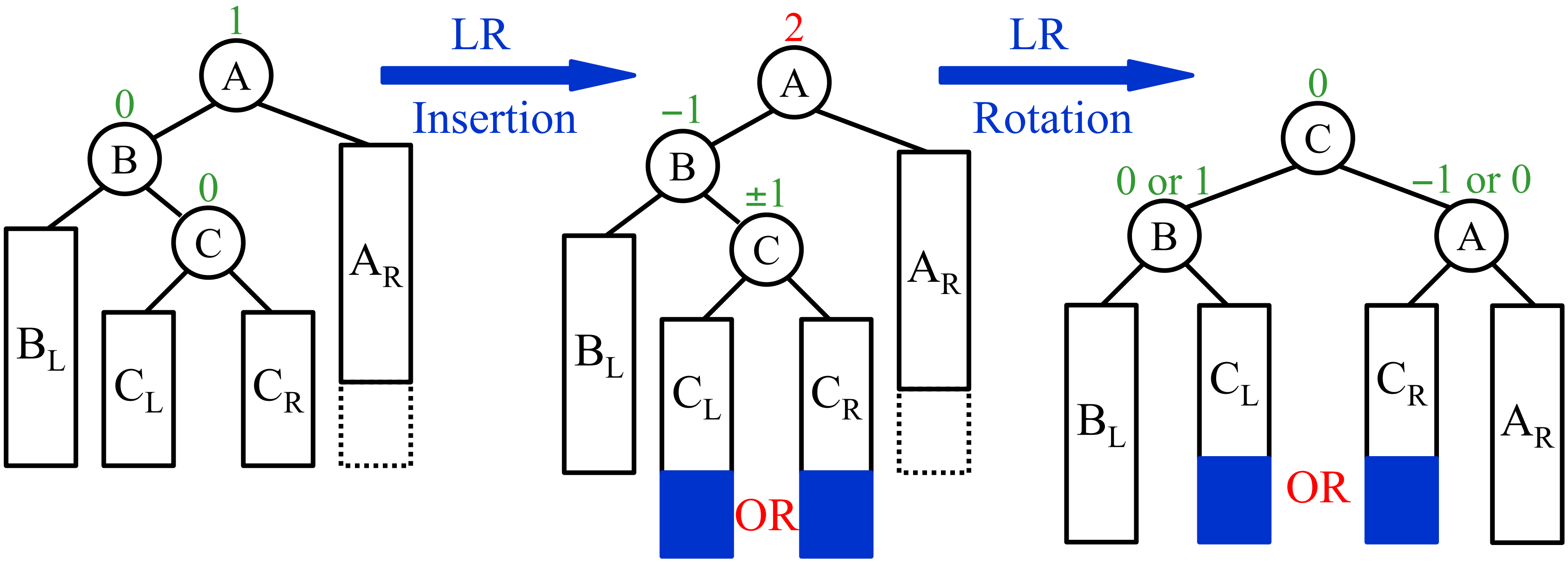

1
A
0
B
0
C
B_L
C_L
C_R
A_R
LR
Insertion
2
A
−1
B
±1
C
B_L
C_L
C_R
A_R
OR



LR

Rotation

In general:

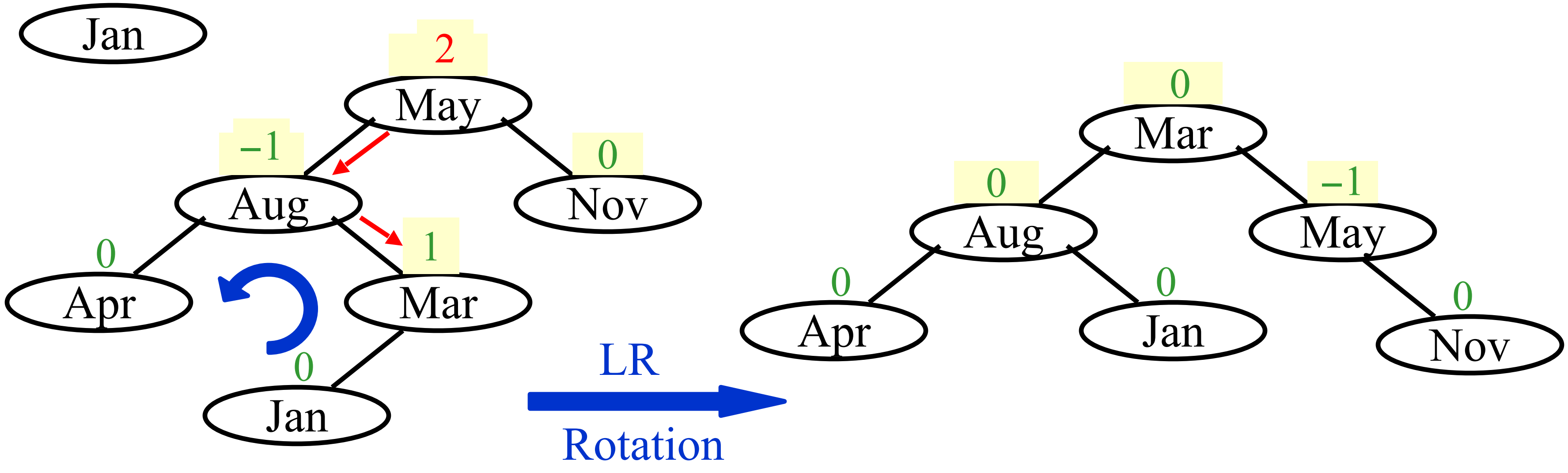


LR

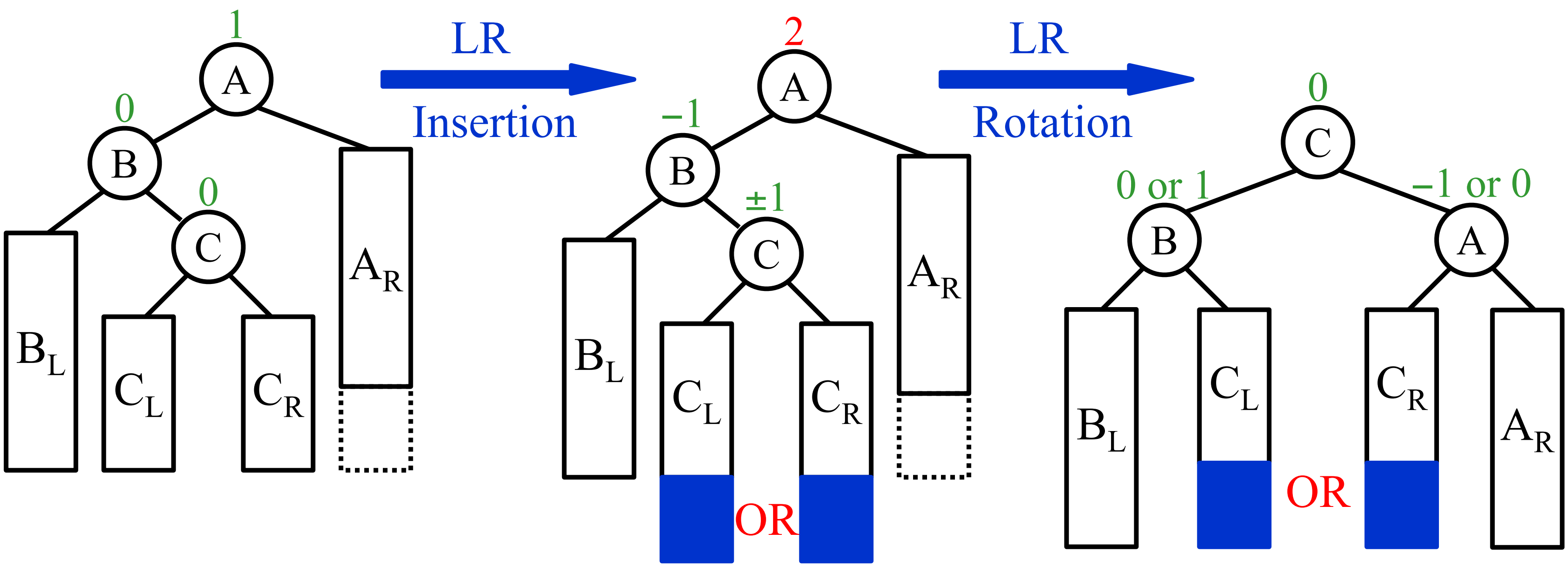
Insertion

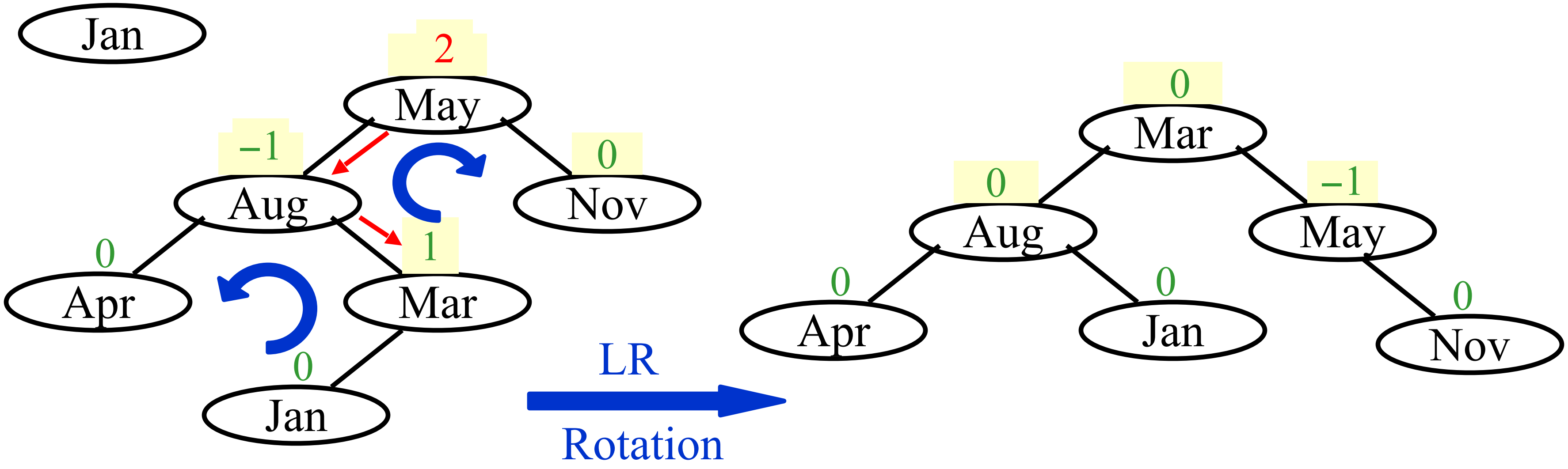
LR

Rotation

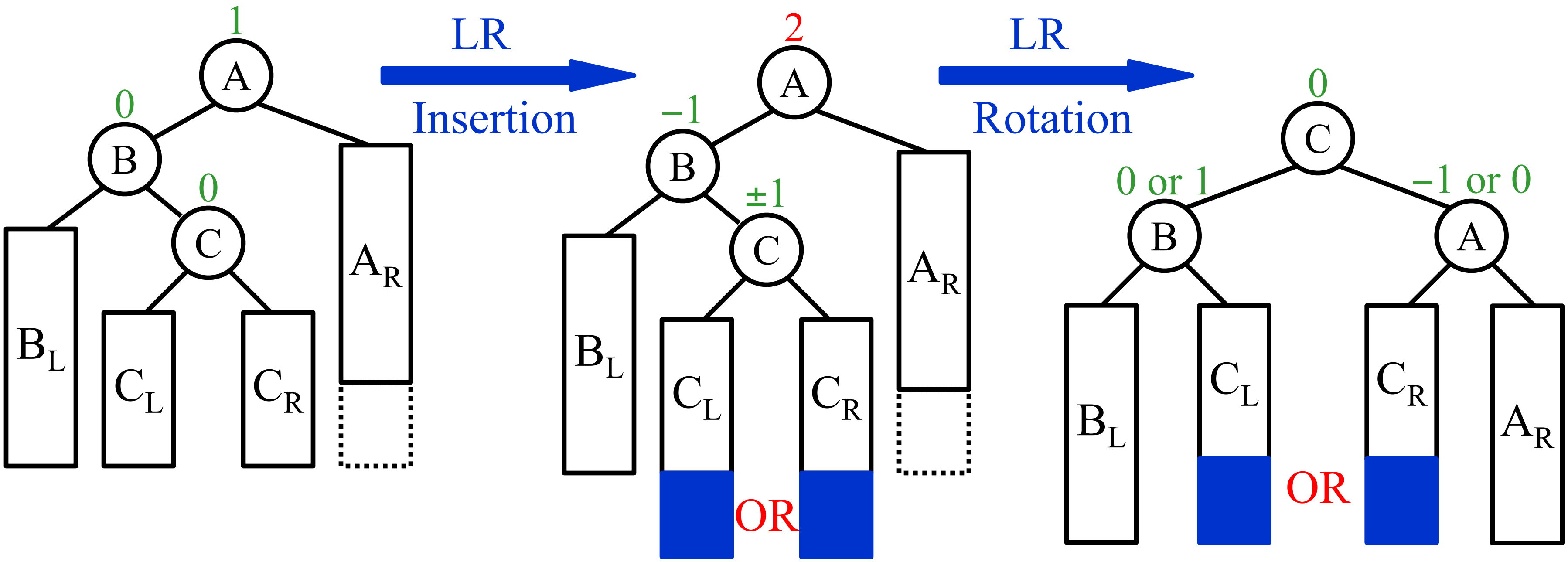

Jan
2
May
−1
Aug
0
Nov
0
Apr
1
Mar
0
Jan
LR
Rotation
0
Mar
0
Aug
−1
May
0
Apr
0
Jan
0
Nov

In general:

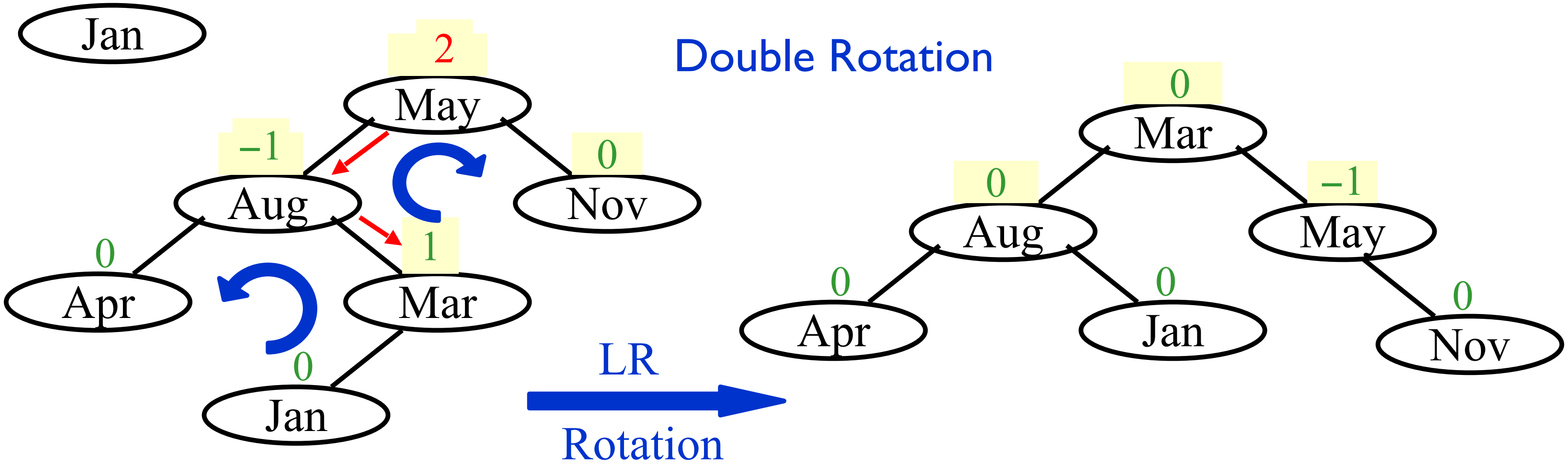

1
A
0
B
0
C
B_L
C_L
C_R
A_R
LR
Insertion
2
A
−1
B
±1
C
B_L
C_L
C_R
A_R
OR
LR
Rotation
0
C
0 or 1
B
−1 or 0
A
B_L
C_L
C_R
A_R
OR


Jan
2
May
−1
Aug
0
Nov
0
Apr
1
Mar
0
Jan
LR
Rotation
0
Mar
0
Aug
−1
May
0
Apr
0
Jan
0
Nov

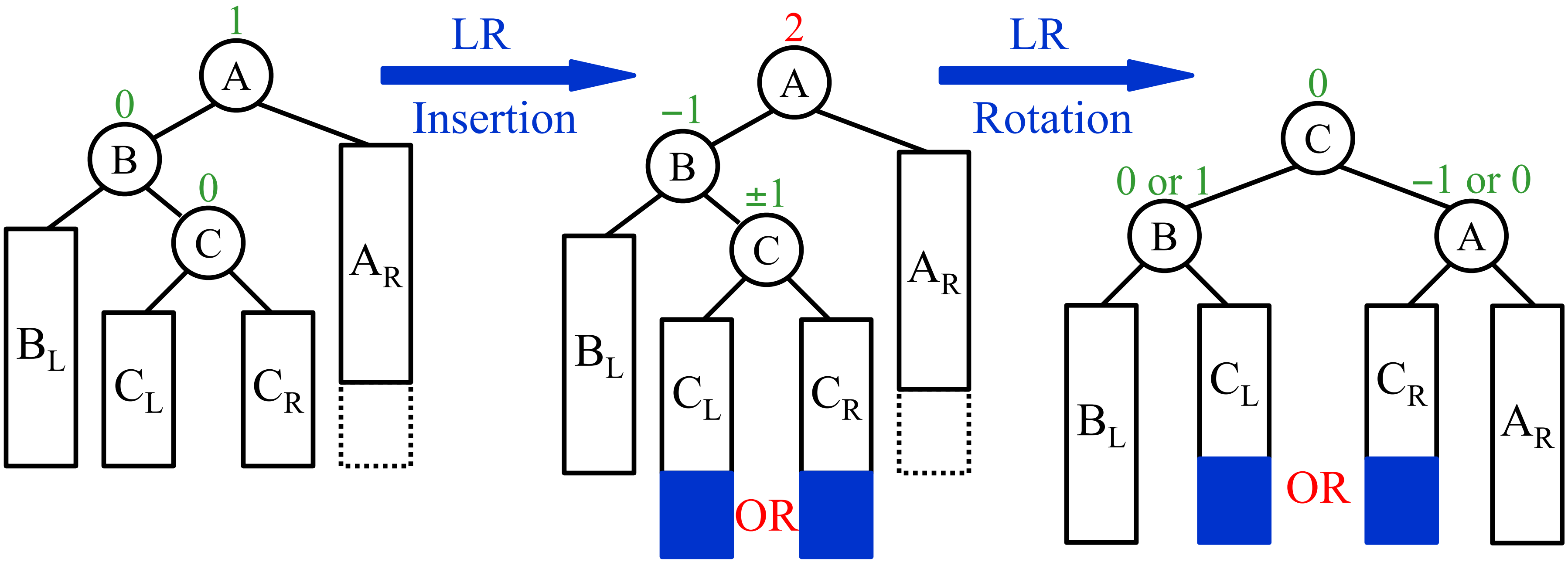
In general:

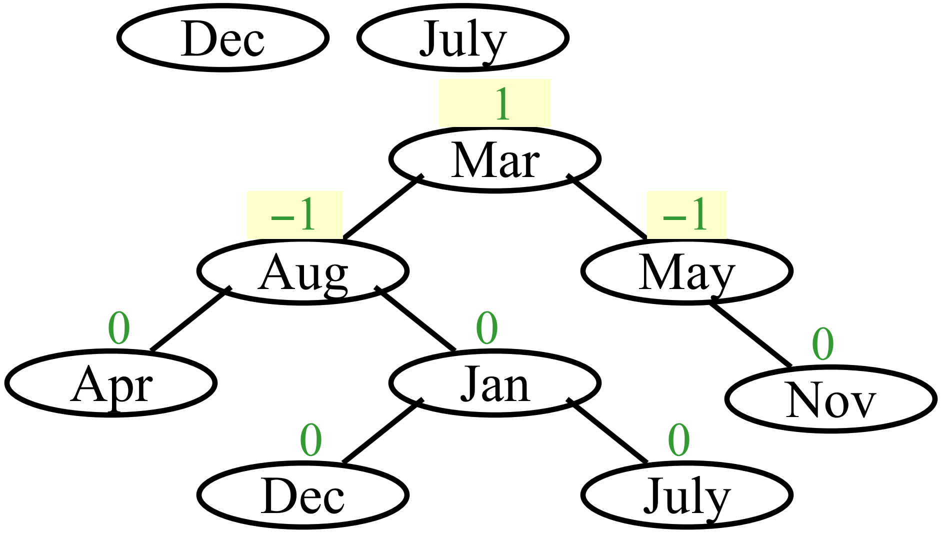

1
A
0
B
0
C
B_L
C_L
C_R
A_R
LR
Insertion
2
A
−1
B
±1
C
B_L
C_L
C_R
A_R
OR
LR
Rotation
0
C
0 or 1
B
−1 or 0
A
B_L
C_L
C_R
A_R
OR

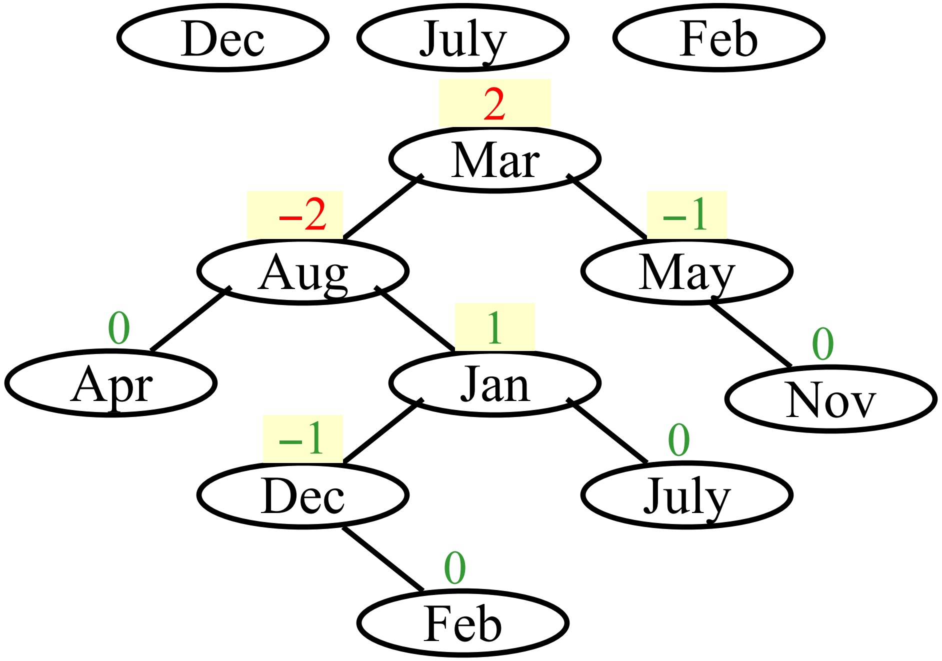
Double Rotation

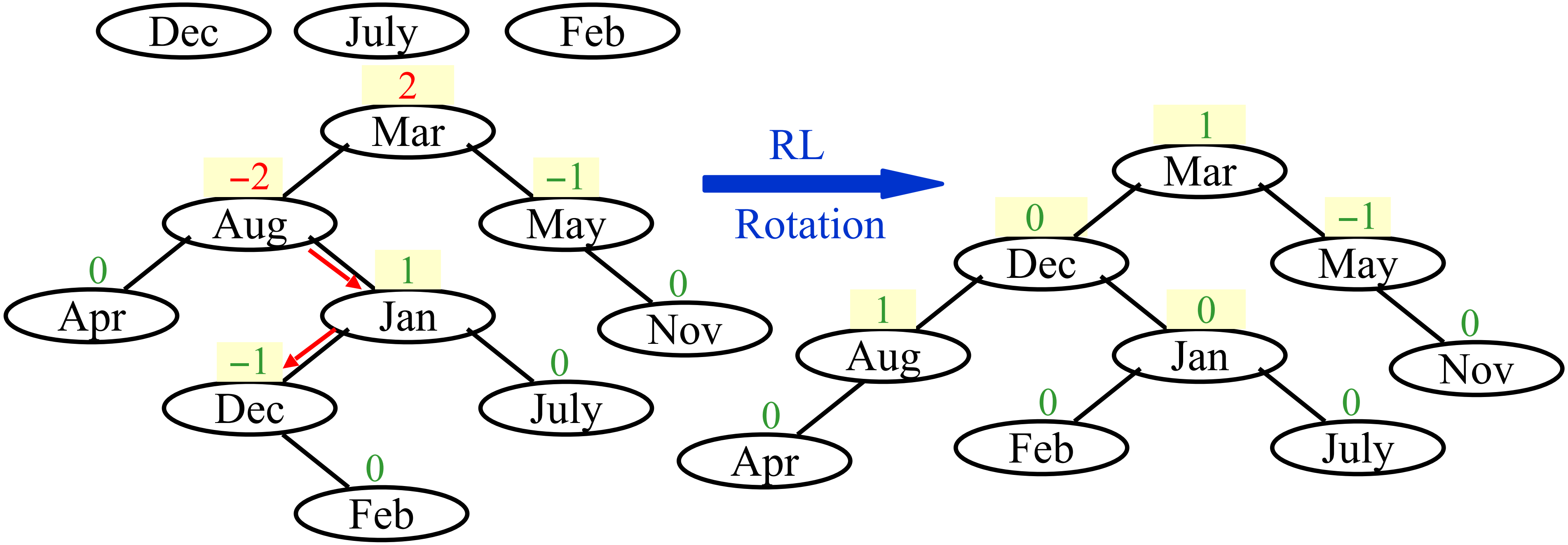


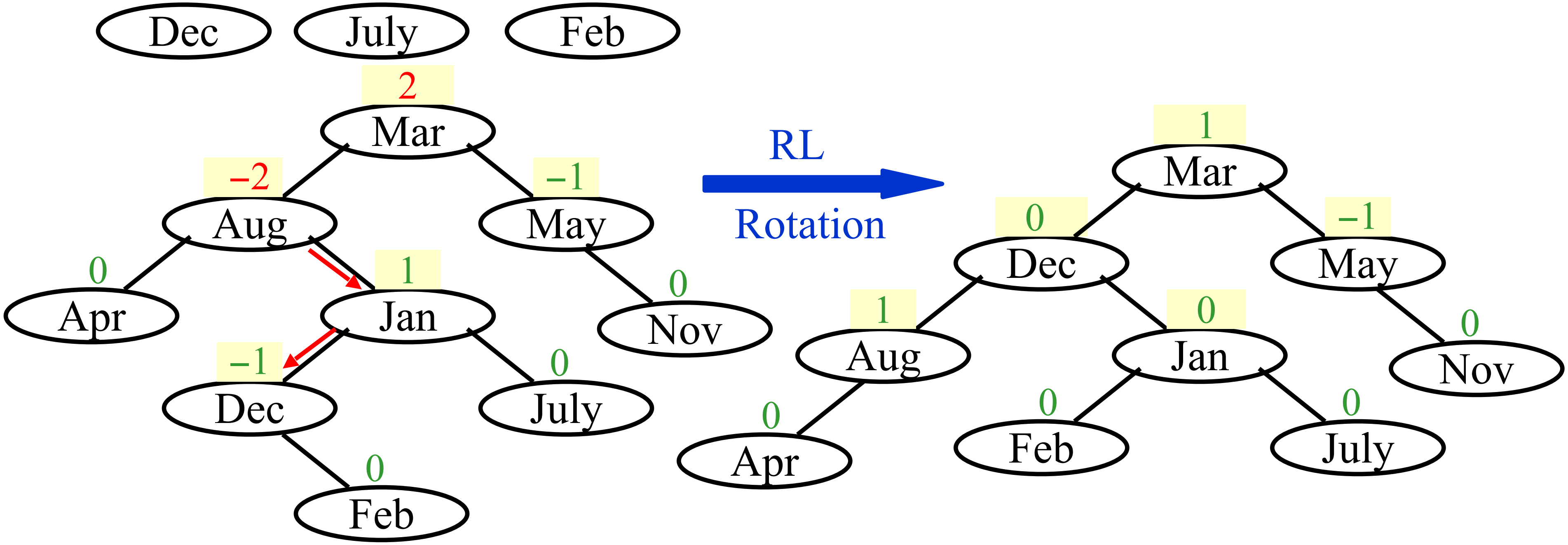
In general:



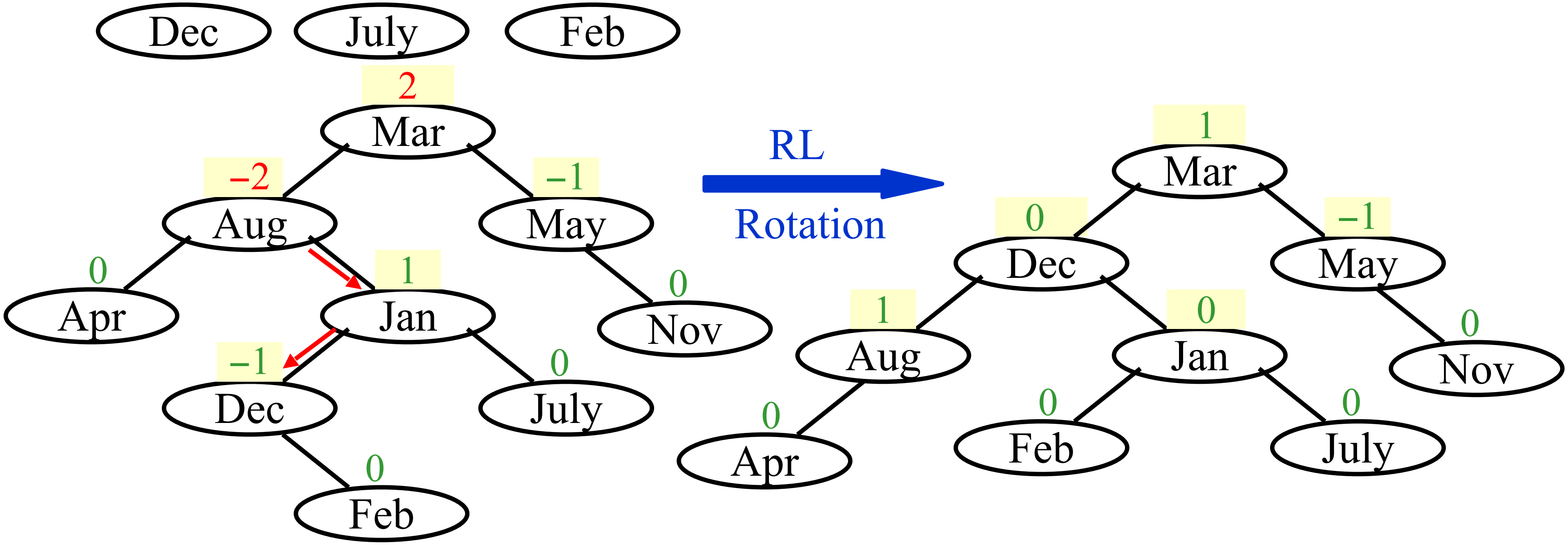

Dec
July
1
Mar
−1
Aug
−1
May
0
Apr
0
Jan
0
Nov
0
Dec
0
July


Dec
July
Feb
2
Mar
–2
Aug
–1
May
0
Apr
1
Jan
0
Nov
–1
Dec
0
July
0
Feb

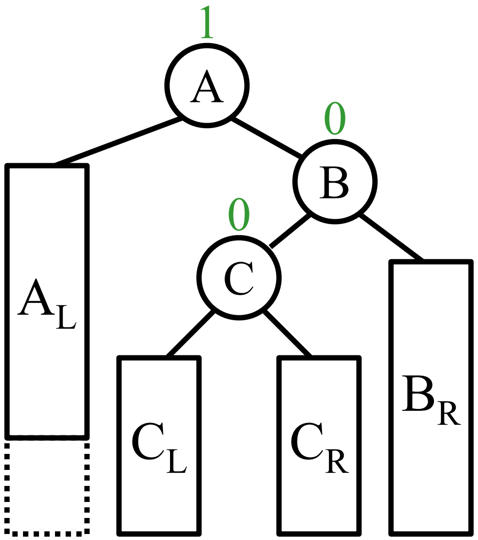

Dec
July
Feb
2
Mar
−2
Aug
−1
May
0
Apr
1
Jan
0
Nov
−1
Dec
0
July
0
Feb
RL
Rotation
1
Mar
0
Dec
−1
May
1
Aug
0
Jan
0
Nov
0
Apr
0
Feb
0
July

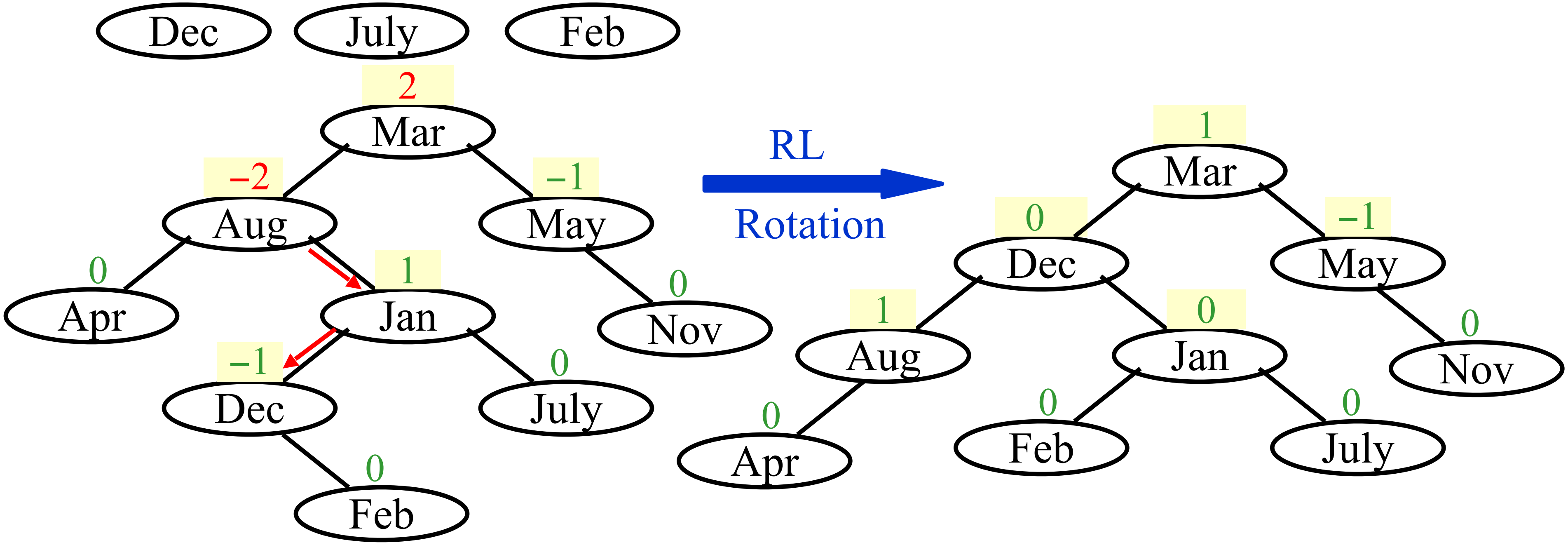

Dec
July
Feb
2
Mar
–2
Aug
–1
May
0
Apr
1
Jan
0
Nov
–1
Dec
0
July
0
Feb
RL
Rotation
1
Mar
0
Dec
–1
May
1
Aug
0
Jan
0
Nov
0
Apr
0
Feb
0
July

In general:


Dec
July
Feb
2
Mar
−2
Aug
−1
May
0
Apr
1
Jan
0
Nov
−1
Dec
0
July
0
Feb
RL
Rotation
1
Mar
0
Dec
−1
May
1
Aug
0
Jan
0
Nov
0
Apr
0
Feb
0
July

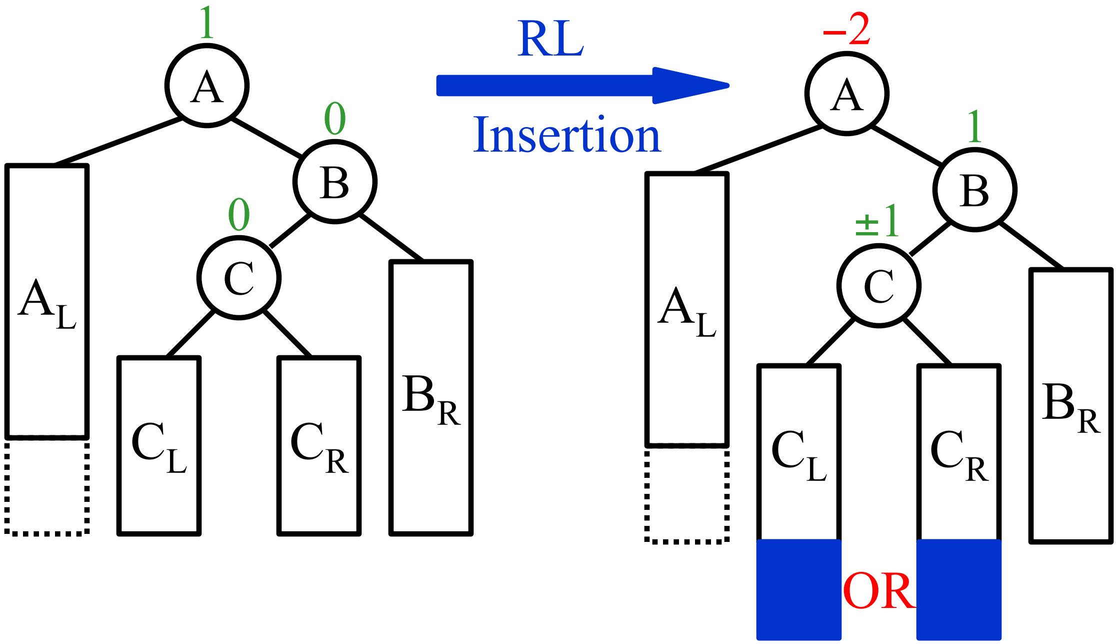
In general:


1
A
0
B
0
C
A_L
C_L
C_R
B_R

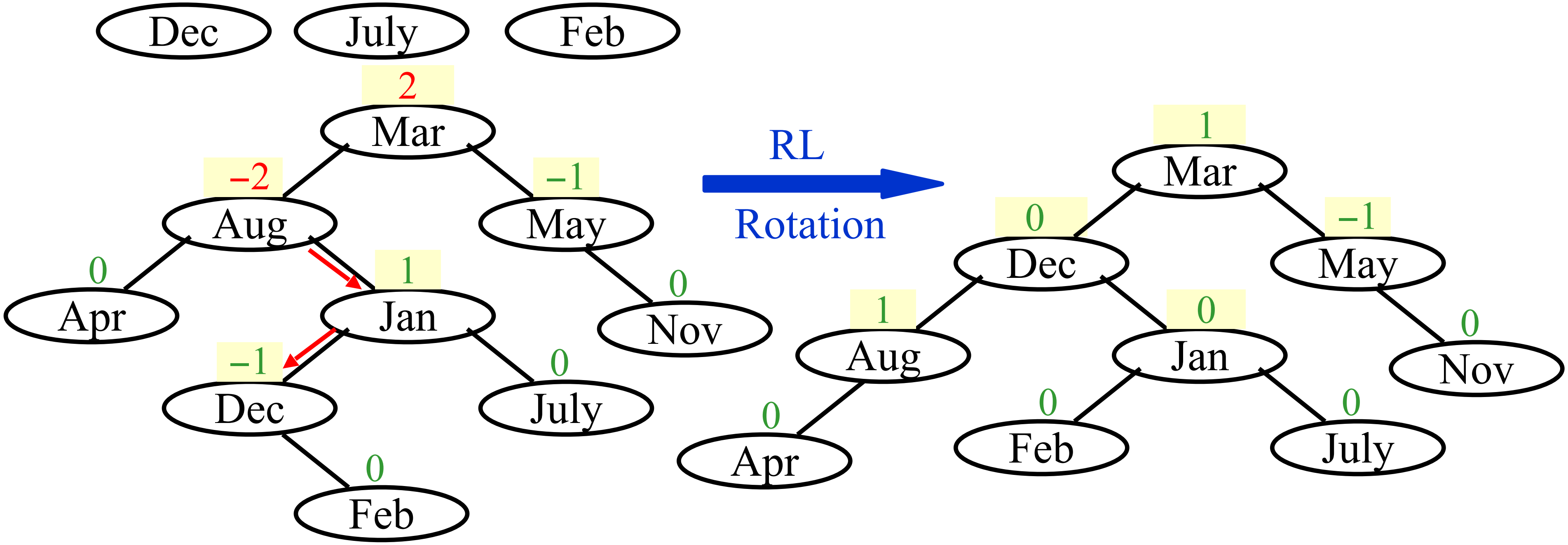


RL Rotation

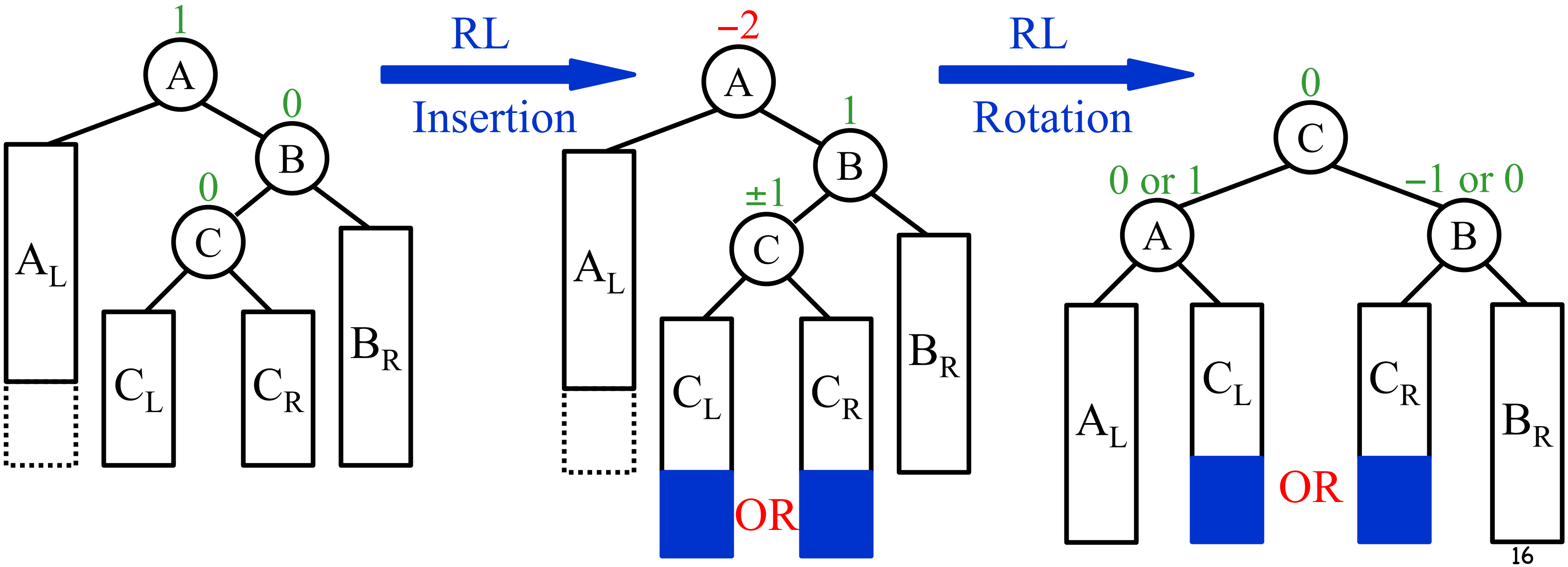
In general:

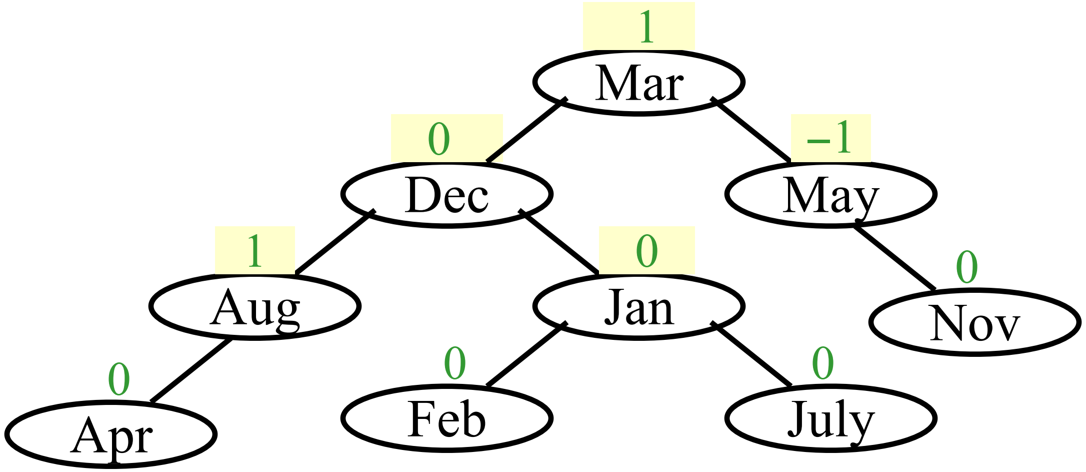


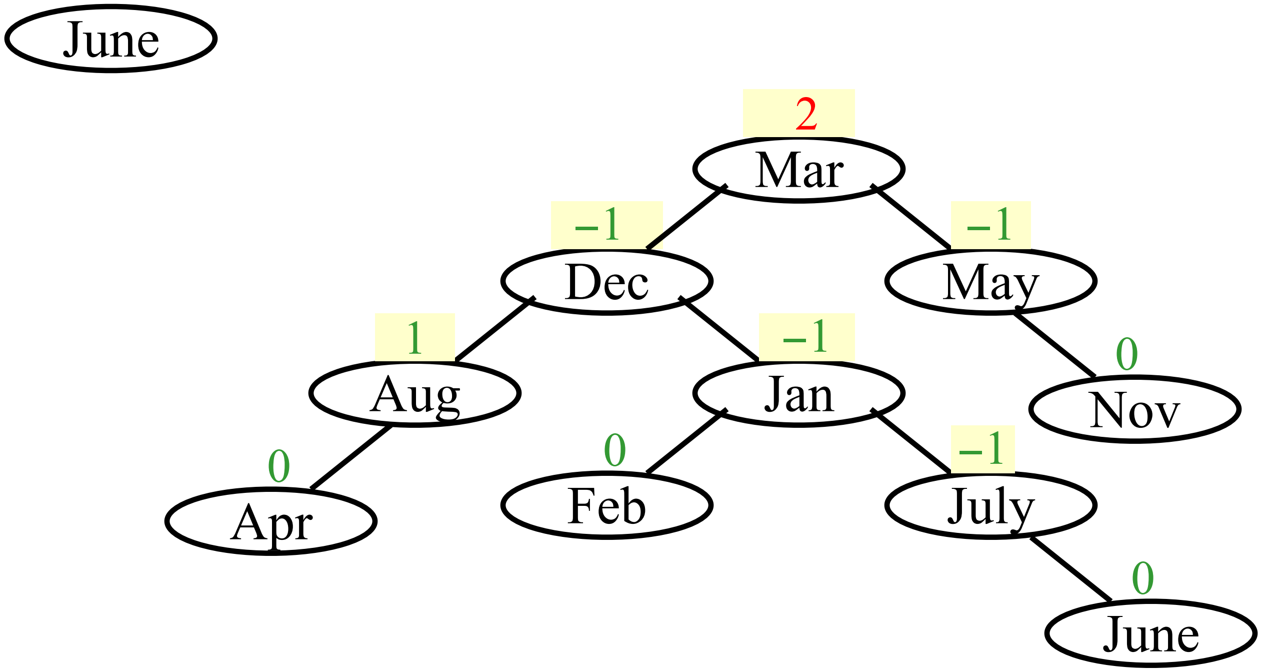
RL Insertion

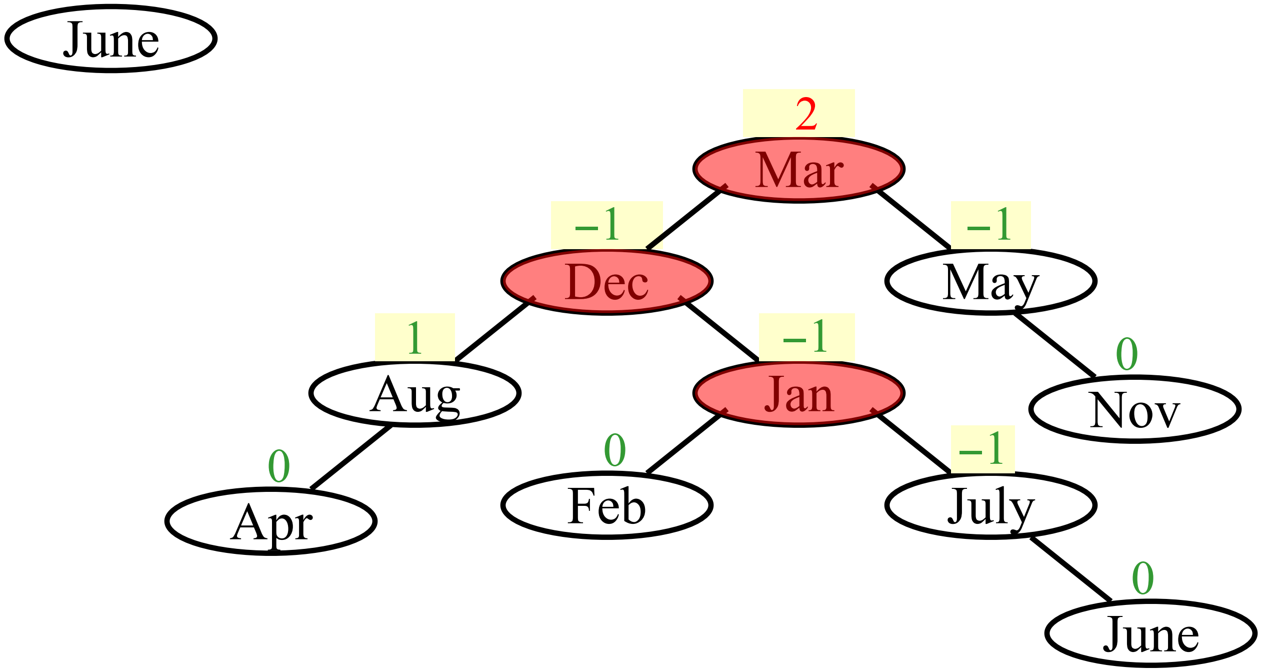

Dec
July
Feb
2
Mar
–2
Aug
–1
May
0
Apr
1
Jan
0
Nov
–1
Dec
0
July
0
Feb
RL
Rotation
1
Mar
0
Dec
–1
May
1
Aug
0
Jan
0
Nov
0
Apr
0
Feb
0
July

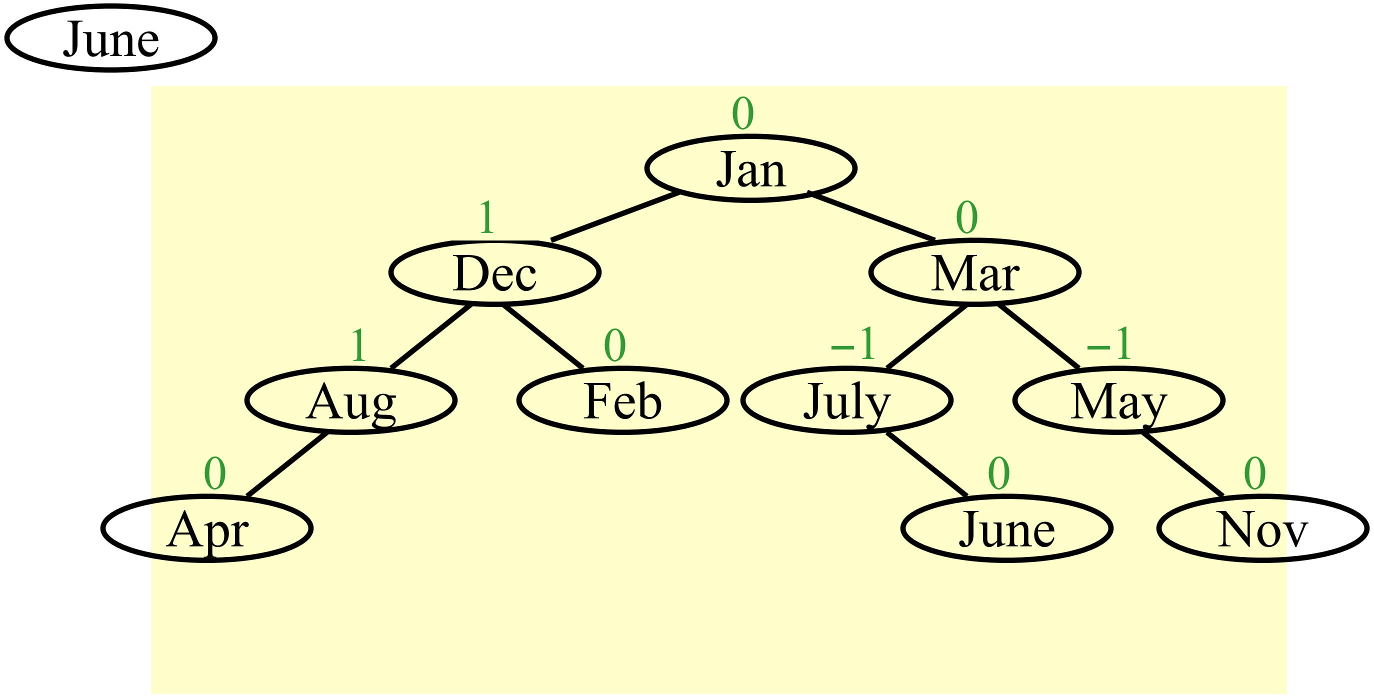
In general:

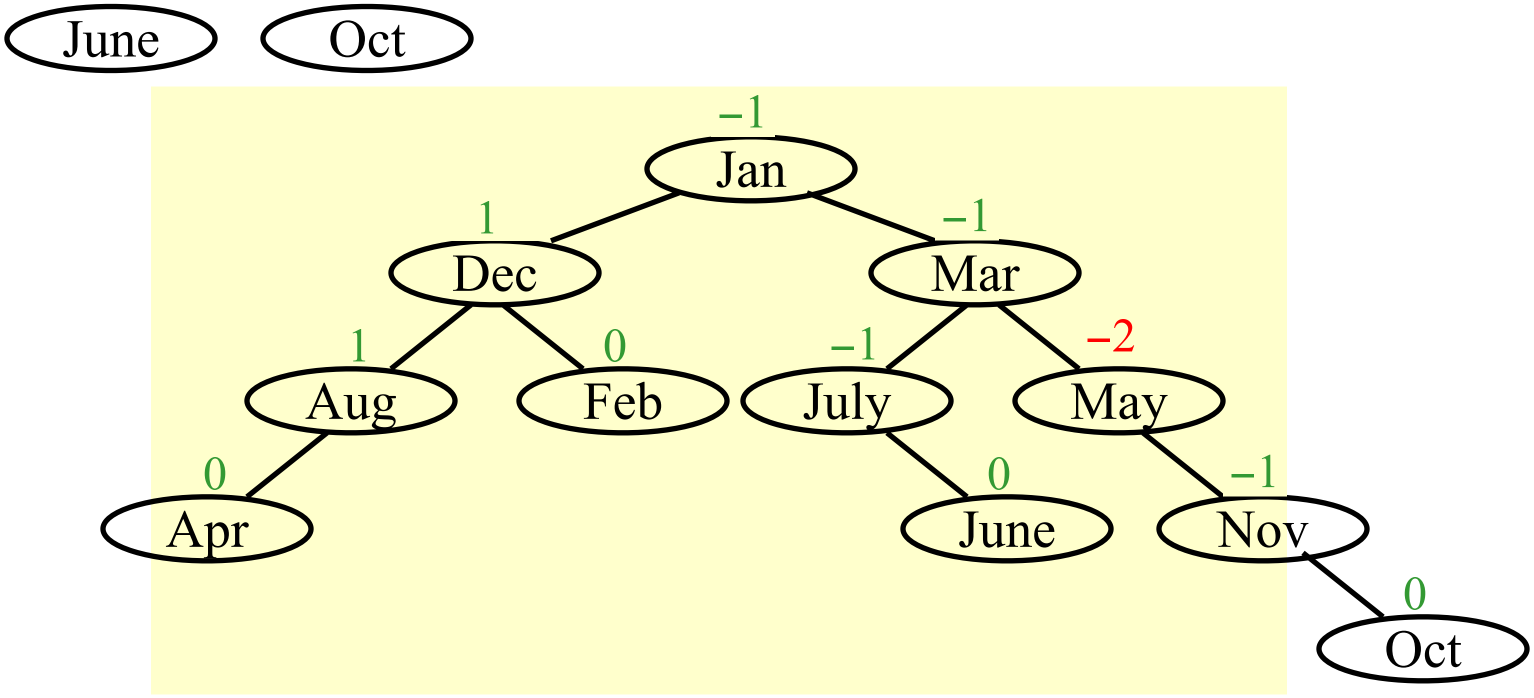

1
A
0
B
0
C
A_L
C_L
C_R
B_R
RL
Insertion
–2
A
1
B
±1
C
A_L
C_L
C_R
B_R
OR
RL
Rotation
0
C
0 or 1
A
–1 or 0
B
A_L
C_L
C_R
B_R
OR

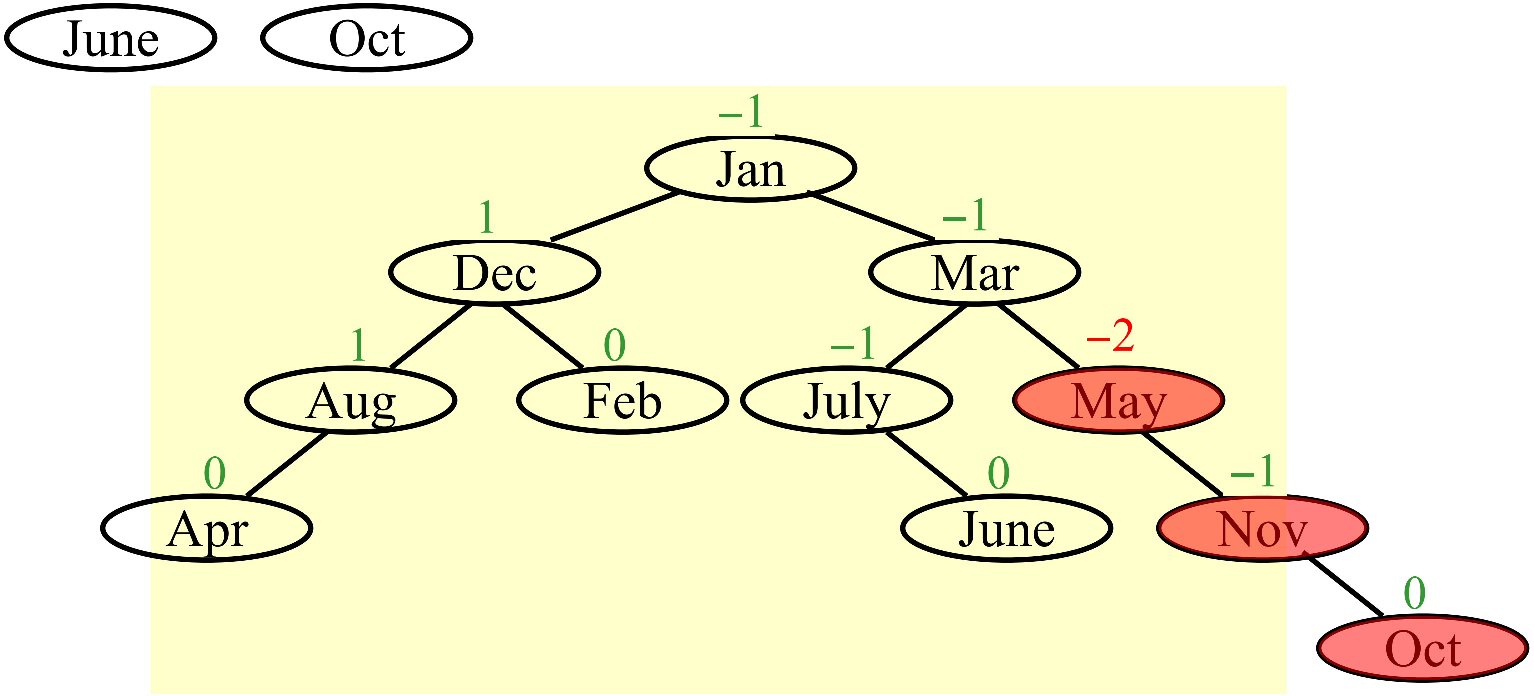

1
Mar
0
Dec
–1
May
1
Aug
0
Jan
0
Nov
0
Apr
0
Feb
0
July

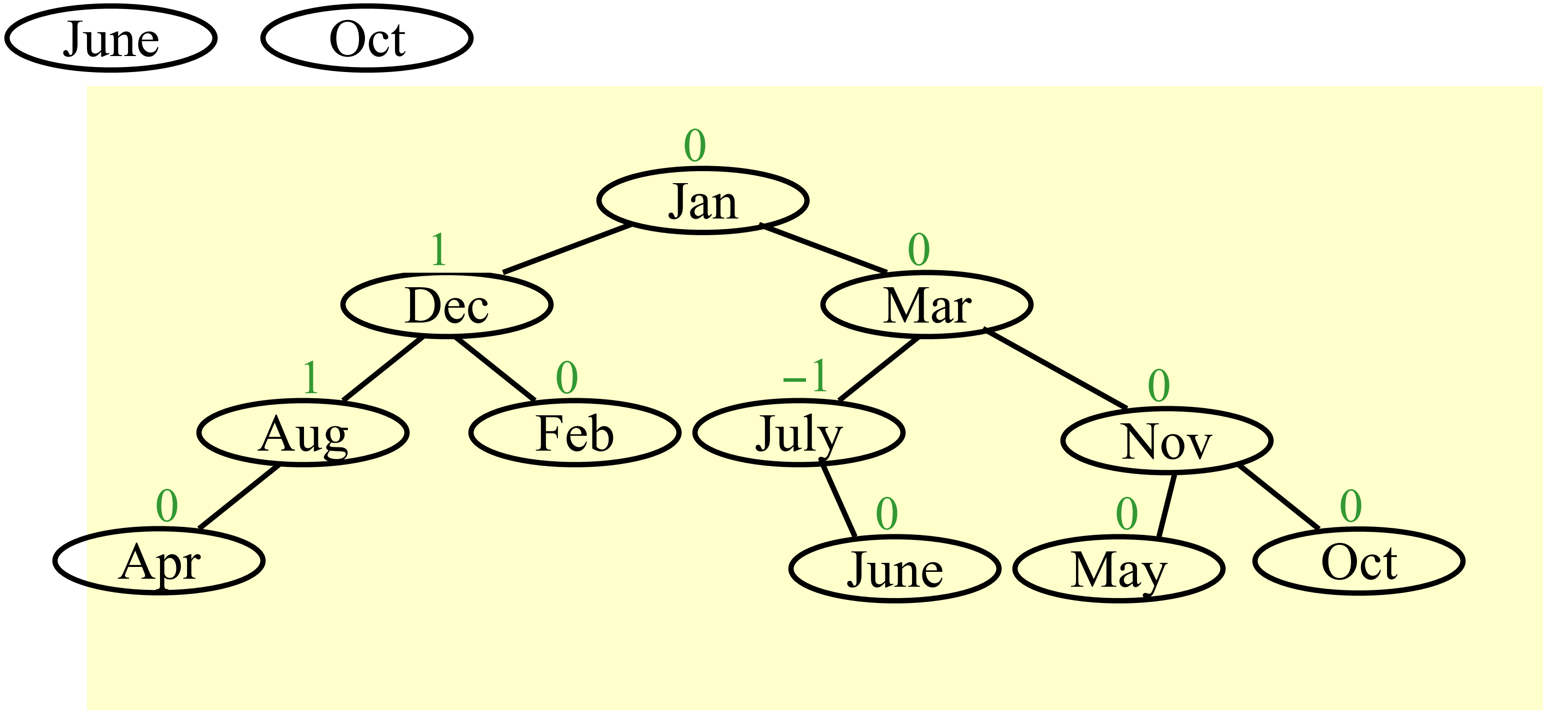

June
2
Mar
–1
Dec
–1
May
1
Aug
–1
Jan
0
Nov
0
Apr
0
Feb
–1
July
0
June

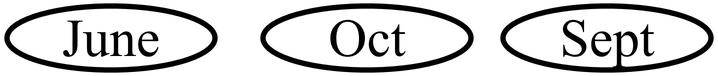

June
2
Mar
−1
Dec
−1
May
1
Aug
−1
Jan
0
Nov
0
Apr
0
Feb
−1
July
0
June

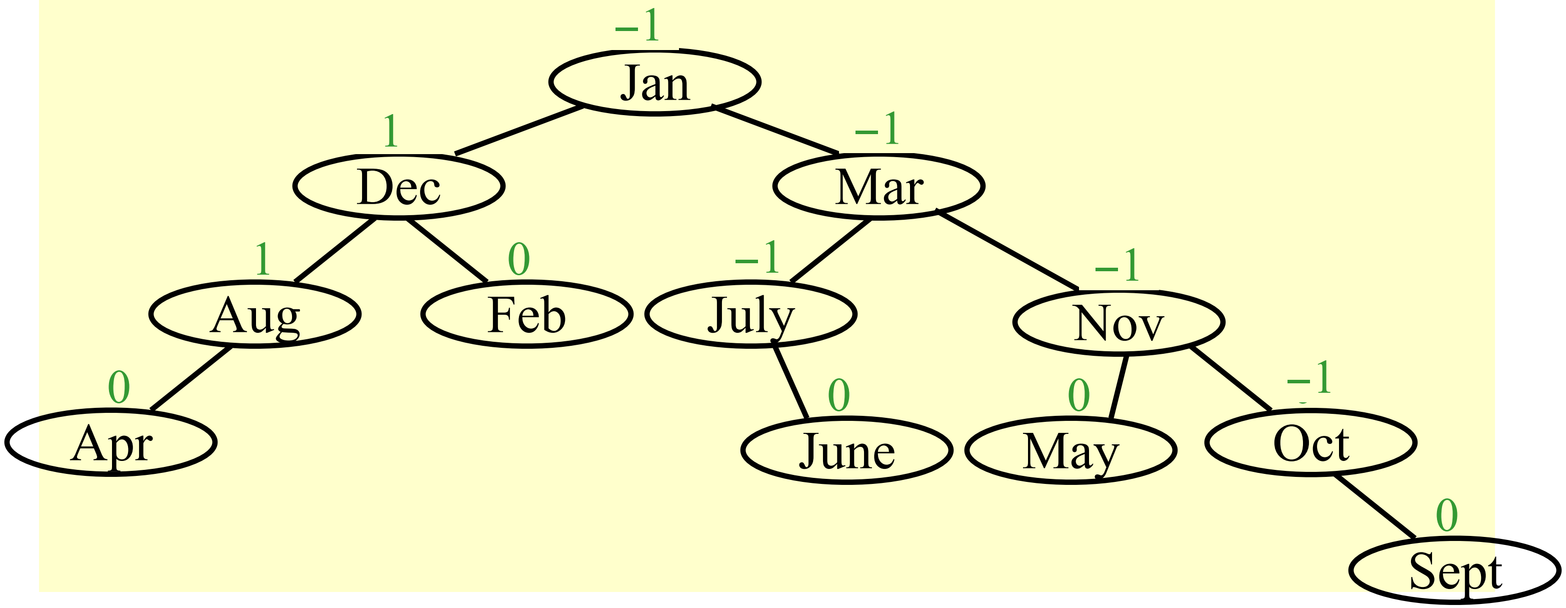

June
0
Jan
1
Dec
0
Mar
1
Aug
0
Feb
–1
July
–1
May
0
Apr
0
June
0
Nov

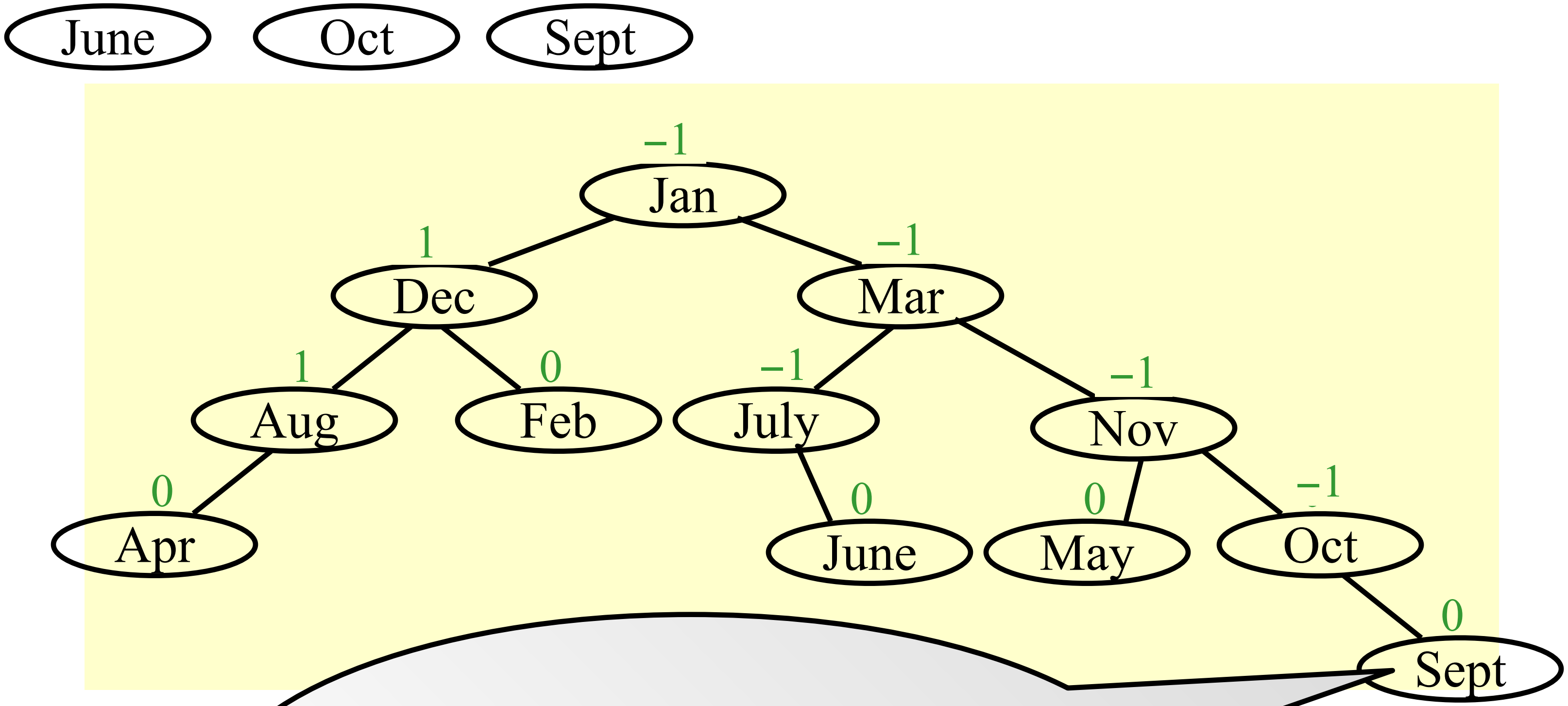

June
Oct
–1
Jan
1
Dec
–1
Mar
1
Aug
0
Feb
–1
July
–2
May
0
Apr
0
June
–1
Nov
0
Oct


June
Oct
–1
Jan
1
Dec
–1
Mar
1
Aug
0
Feb
–1
July
–2
May
0
Apr
0
June
–1
Nov
0
Oct

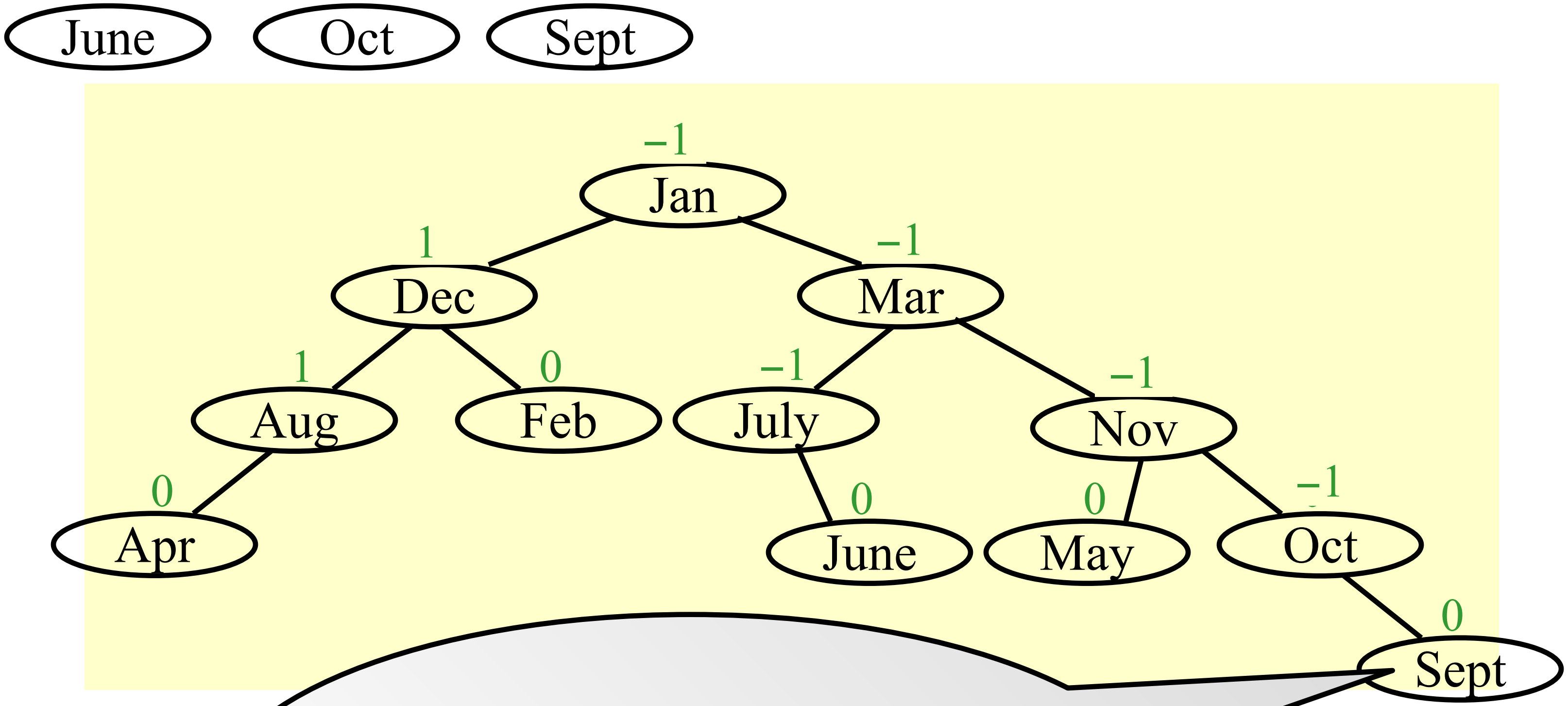

June
Oct
0
Jan
1
Dec
0
Mar
1
Aug
0
Feb
−1
July
0
Nov
0
Apr
0
June
0
May
0
Oct


June
Oct
Sept


−1
Jan
1
Dec
−1
Mar
1
Aug
0
Feb
−1
July
−1
Nov
0
Apr
0
June
0
May
−1
Oct
0
Sept

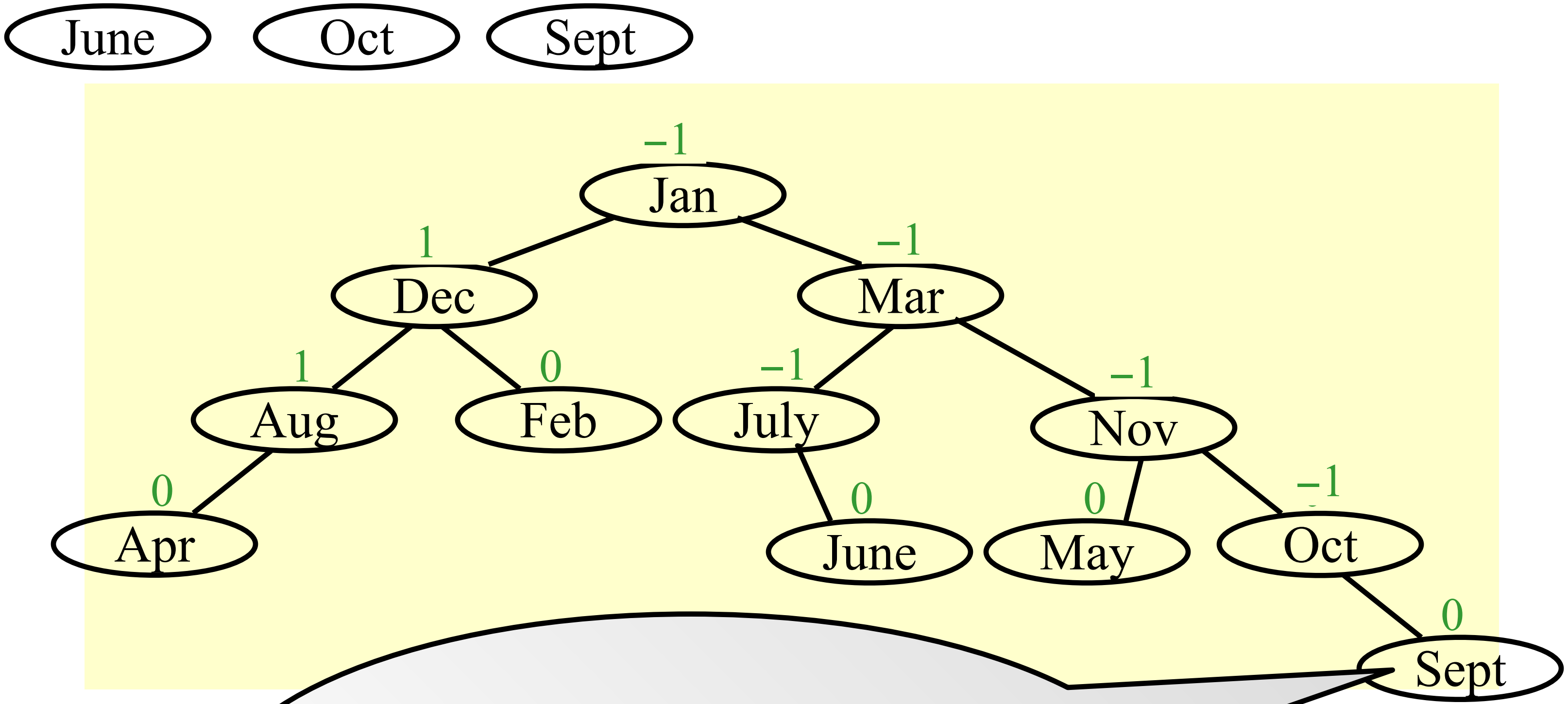


Note: Several bf's
might be changed even if
we don't need to reconstruct
the tree.



Note: Several bf's might be changed even if we don't need to reconstruct the tree.

Another option is to keep a *height* field for each node.

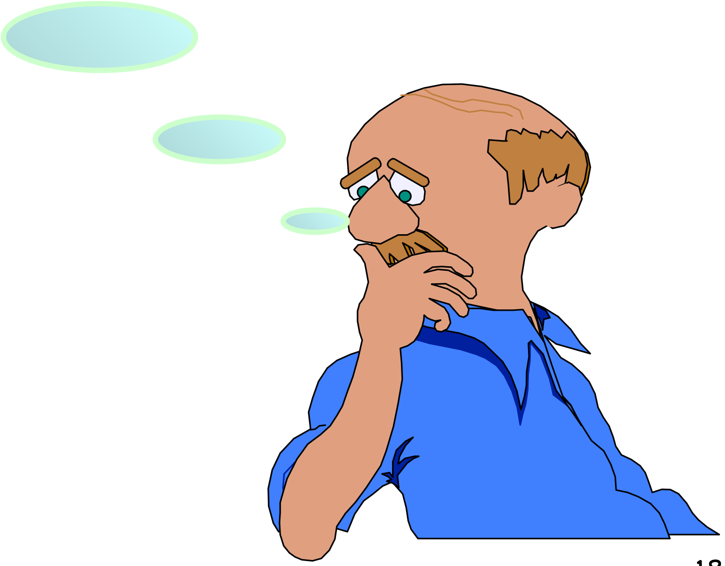


Note: Several bf's might be changed even if we don't need to reconstruct the tree.

Another option is to keep a *height* field for each node.

Read the declaration and functions in [Weiss] Figures 4.42 – 4.48

One last question:
Obviously we have T_p = O(h)
where h is the height of the tree.
But h = ?

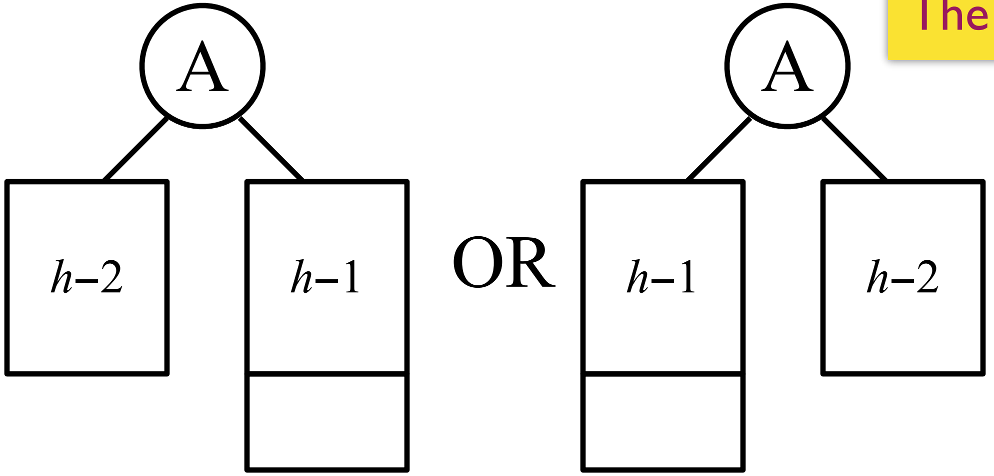


Let n_h be the minimum number of nodes in a height-balanced tree of height h. What does the tree look like?

The worst case for AVL tree of height h.

Let n_h be the minimum number of nodes in a height-balanced tree of height h. What does the tree look like?

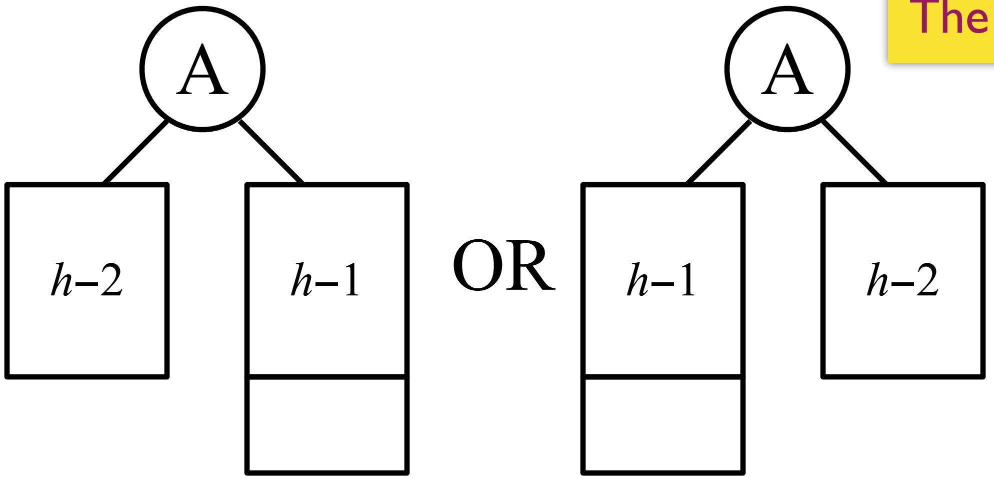
The worst case for AVL tree of height h.



$$\Rightarrow \quad n_h = n_{h-1} + n_{h-2} + 1$$

Let n_h be the minimum number of nodes in a height-balanced tree of height h. What does the tree look like?

The worst case for AVL tree of height h.



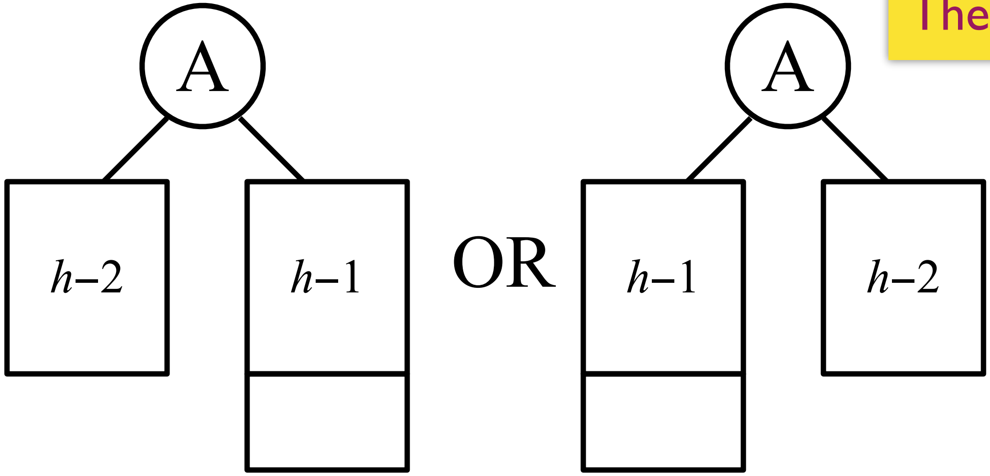
$$\Rightarrow \quad n_h = n_{h-1} + n_{h-2} + 1$$

Fibonacci numbers:

$$F_0 = 0,\ F_1 = 1,\ F_i = F_{i-1} + F_{i-2} \quad \text{for } \ i > 1$$

Let n_h be the minimum number of nodes in a height-balanced tree of height h. What does the tree look like?

The worst case for AVL tree of height h.



$$\Rightarrow \quad n_h = n_{h-1} + n_{h-2} + 1$$

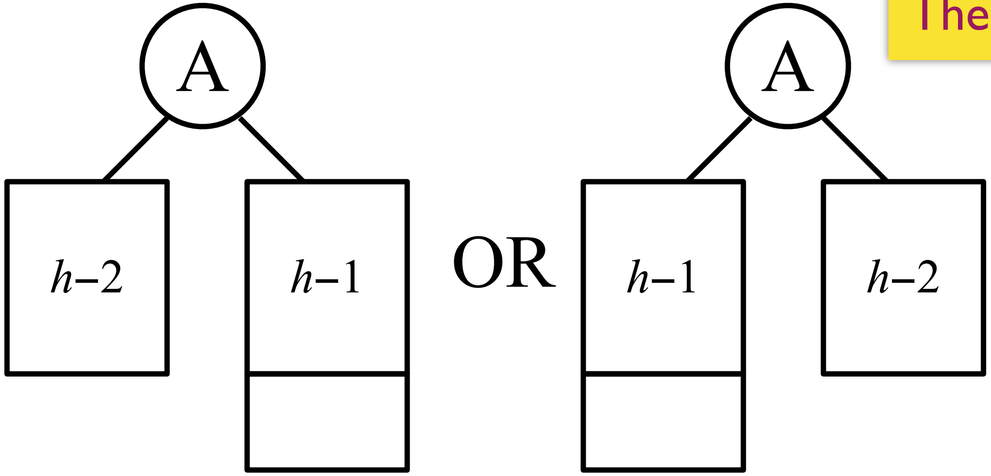
Fibonacci numbers:

$$F_0 = 0,\ F_1 = 1,\ F_i = F_{i-1} + F_{i-2} \quad \text{for} \ \ i > 1$$

$$\Rightarrow \quad n_h = F_{h+3} - 1, \ \text{ for } \ h \geq 0$$

Let n_h be the minimum number of nodes in a height-balanced tree of height h. What does the tree look like?

The worst case for AVL tree of height h.



$$\Rightarrow \quad n_h = n_{h-1} + n_{h-2} + 1$$

Fibonacci numbers:

$$F_0 = 0,\ F_1 = 1,\ F_i = F_{i-1} + F_{i-2} \quad \text{for} \quad i > 1$$

$$\Rightarrow \quad n_h = F_{h+3} - 1, \ \text{for} \quad h \geq 0$$

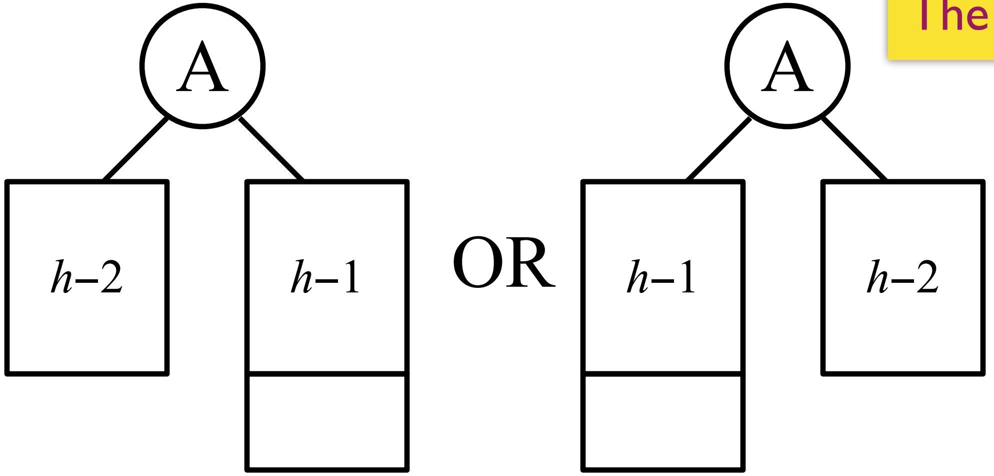
Fibonacci number theory gives that

$$F_i \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^i$$

3

Let n_h be the minimum number of nodes in a height-balanced tree of height h. What does the tree look like?

The worst case for AVL tree of height h.



$$\Rightarrow \quad n_h = n_{h-1} + n_{h-2} + 1$$

Fibonacci numbers:

$$F_0 = 0,\ F_1 = 1,\ F_i = F_{i-1} + F_{i-2} \quad \text{for } \ i > 1$$

$$\Rightarrow \quad n_h = F_{h+3} - 1, \ \text{for } \ h \geq 0$$

Fibonacci number theory gives that

$$F_i \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^i$$

$$\Rightarrow \quad n_h \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+3} - 1 \qquad \Rightarrow \quad h = O(\ln n)$$

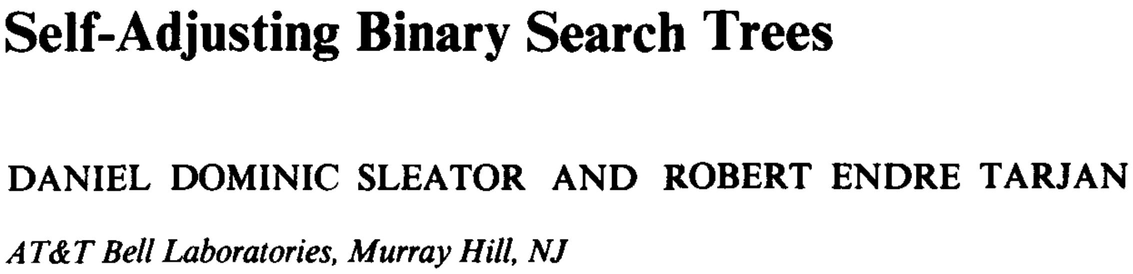
Outline: Balanced Binary Search Trees (1)

- Binary search trees
- AVL trees
- Splay trees
- Amortized analysis
- Take-home messages

Splay Trees (1985)

Daniel Sleator

Robert Tarjan



Self-Adjusting Binary Search Trees

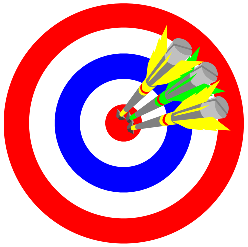
DANIEL DOMINIC SLEATOR AND ROBERT ENDRE TARJAN

AT&T Bell Laboratories, Murray Hill, NJ

Figure courtesy: https://csd.cmu.edu/people/faculty/daniel-sleator
https://en.wikipedia.org/wiki/Robert_Tarjan

Splay Trees

Splay Trees

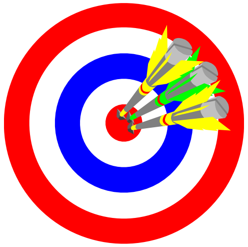


Target : Any M consecutive tree operations starting from an empty tree take at most $O(M \log N)$ time.

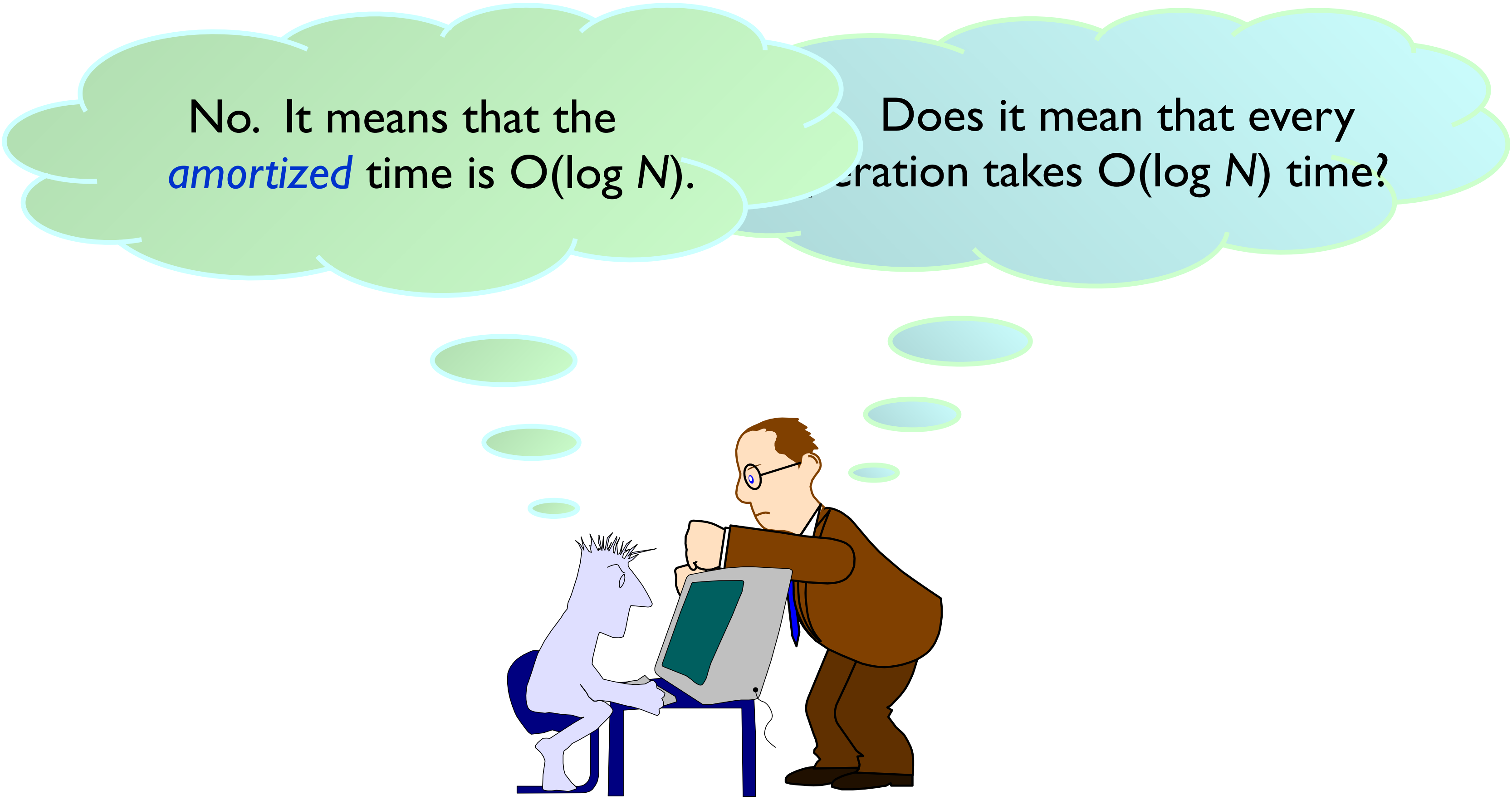
Splay Trees

Target : Any *M* consecutive tree operations starting from an empty tree take at most O(*M* log *N*) time.

Splay Trees



Target : Any M consecutive tree operations starting from an empty tree take at most O(M log N) time.



Does it mean that every eration takes O(log N) time?

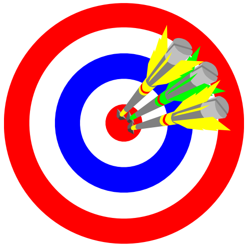
No. It means that the *amortized* time is O(log N).

Splay Trees

Target : Any M consecutive tree operations starting from an empty tree take at most O(M log N) time.

So a single operation might still take O(N) time? Then what's the point?
No. It means that the *amortized* time is O(log N).

Splay Trees



Target : Any *M* consecutive tree operations starting from an empty tree take at most O(*M* log *N*) time.

The bound is weaker.
But the effect is the same:
There are no bad input sequences.

o a single operation might
still take O(*N*) time?
en what's the point?

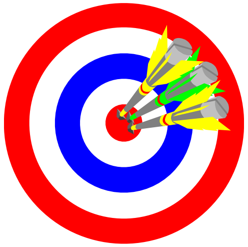
Splay Trees

Target : Any *M* consecutive tree operations starting from an empty tree take at most O(*M* log *N*) time.

The bound is weaker.
But the effect is the sam
There are no bad input seque

But if one node takes O(*N*) time to access, we can keep accessing it for *M* times, can't we?

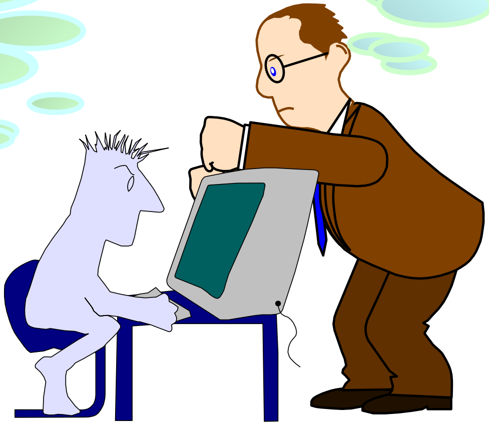
Splay Trees



Target : Any *M* consecutive tree operations starting from an empty tree take at most O(*M* log *N*) time.

Surely we can – that only means that whenever a node is accessed, it must be moved. Otherwise visiting a bad node repeatedly leads to bad performance

t if one node takes O(*N*) time
ess, we can keep accessing it
or *M* times, can't we?



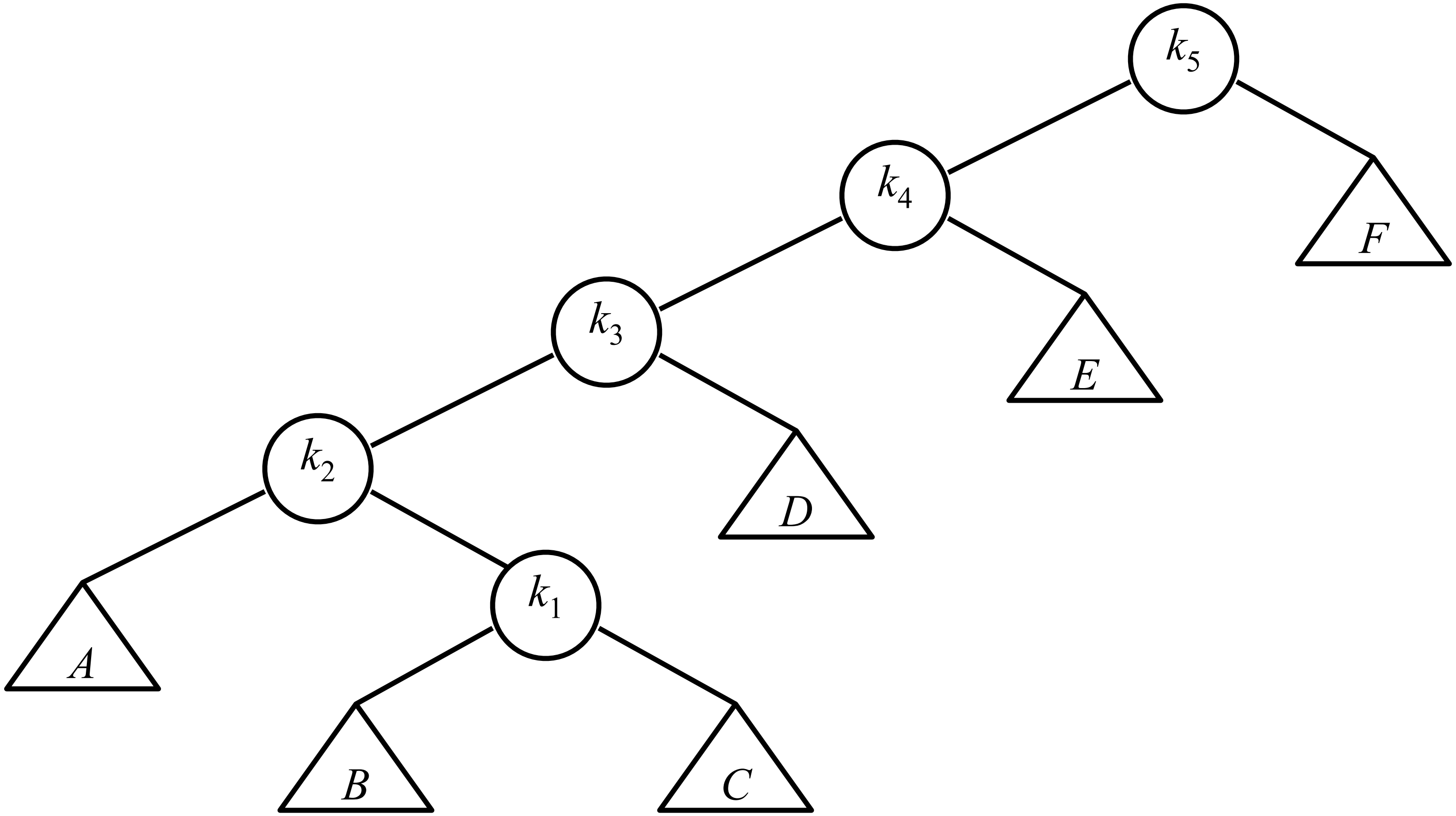
Splay Trees

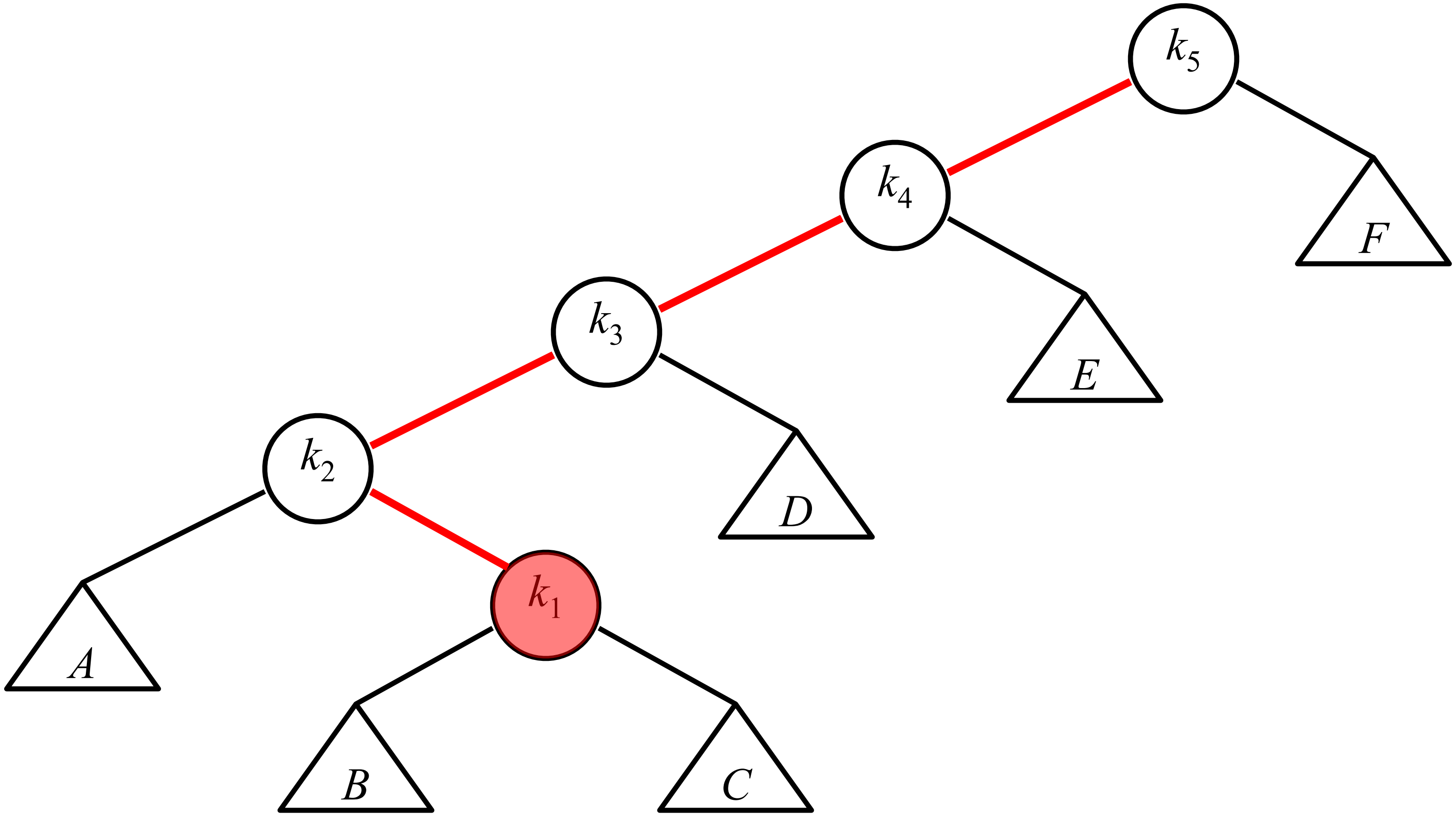
Target : Any *M* consecutive tree operations starting from an empty tree take at most O(*M* log *N*) time.

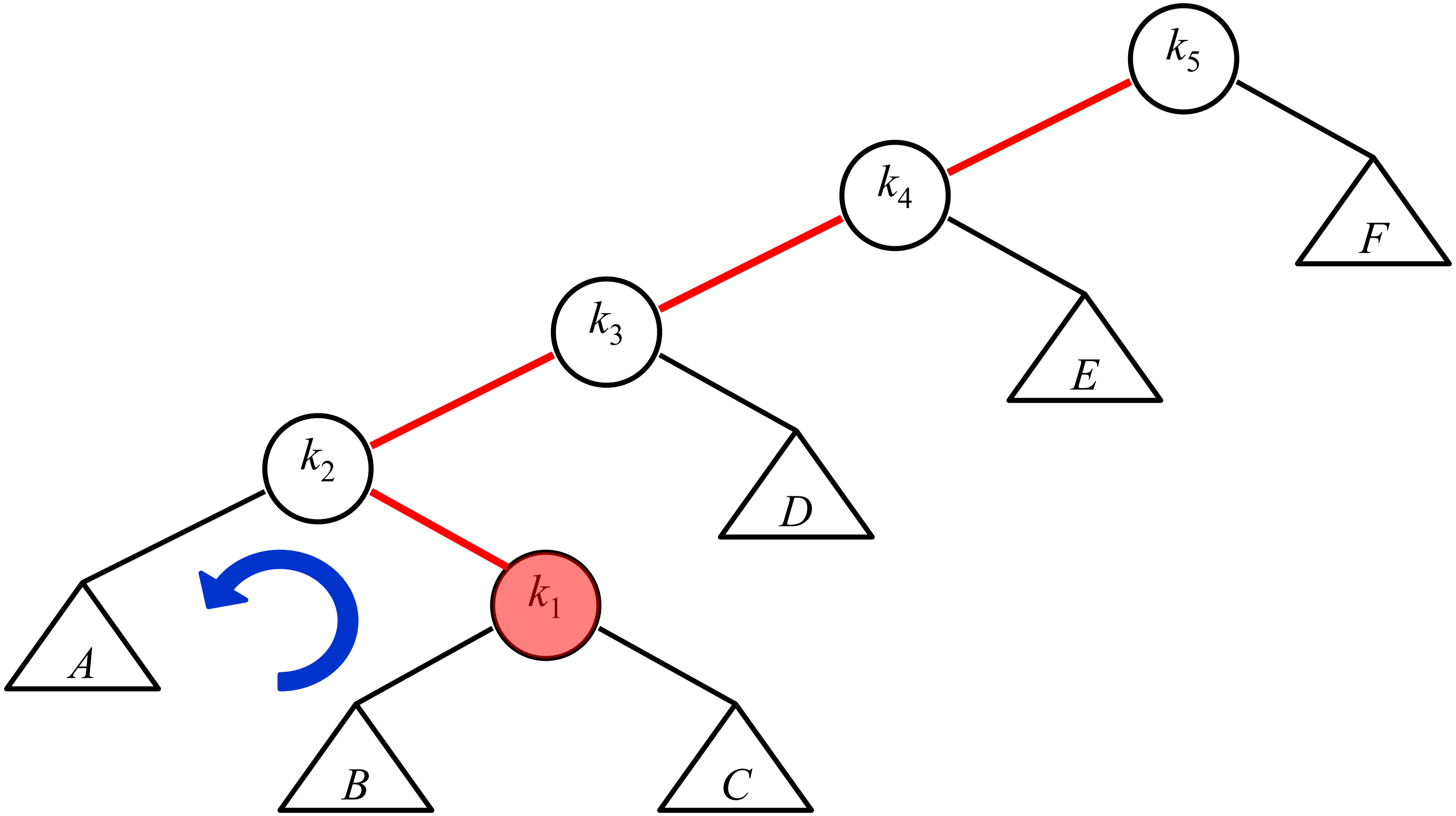
t if one node takes O(*N*) time
ess, we can keep accessing it
or *M* times, can't we?

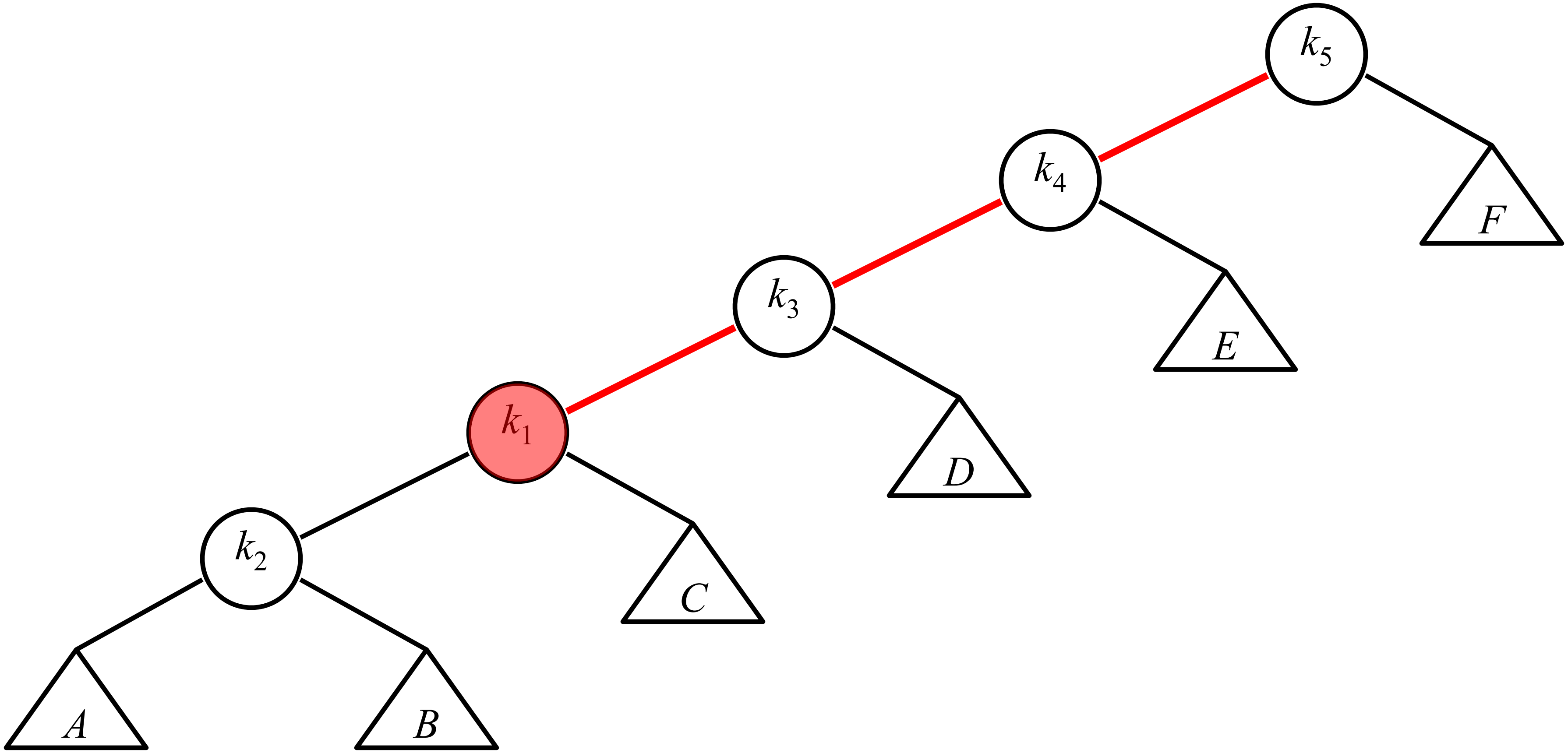
Surely we can – that only means
that whenever a node is accessed,
it must be moved. Otherwise visiting a
bad node repeatedly leads to bad performance

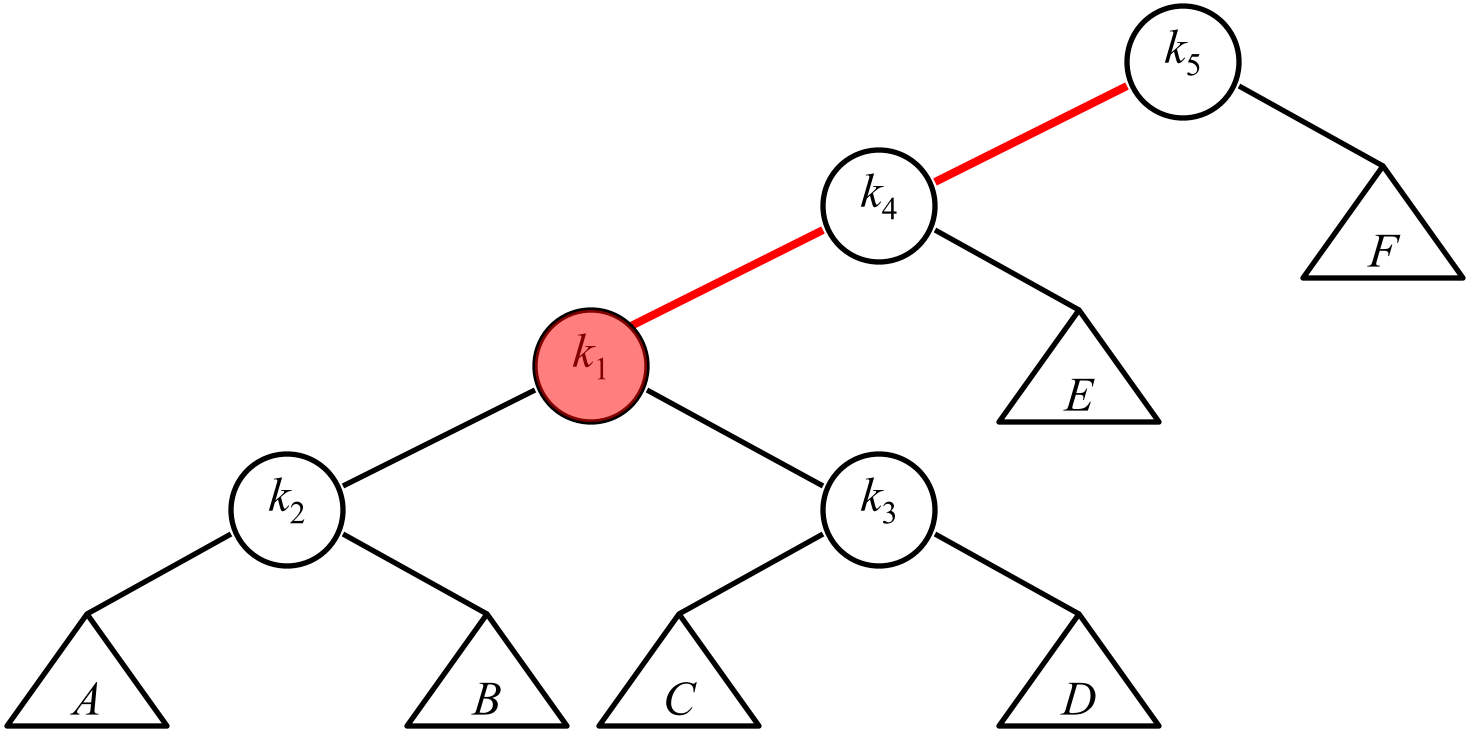
Idea : After a node is accessed, it is pushed to the root by a series of AVL tree rotations.

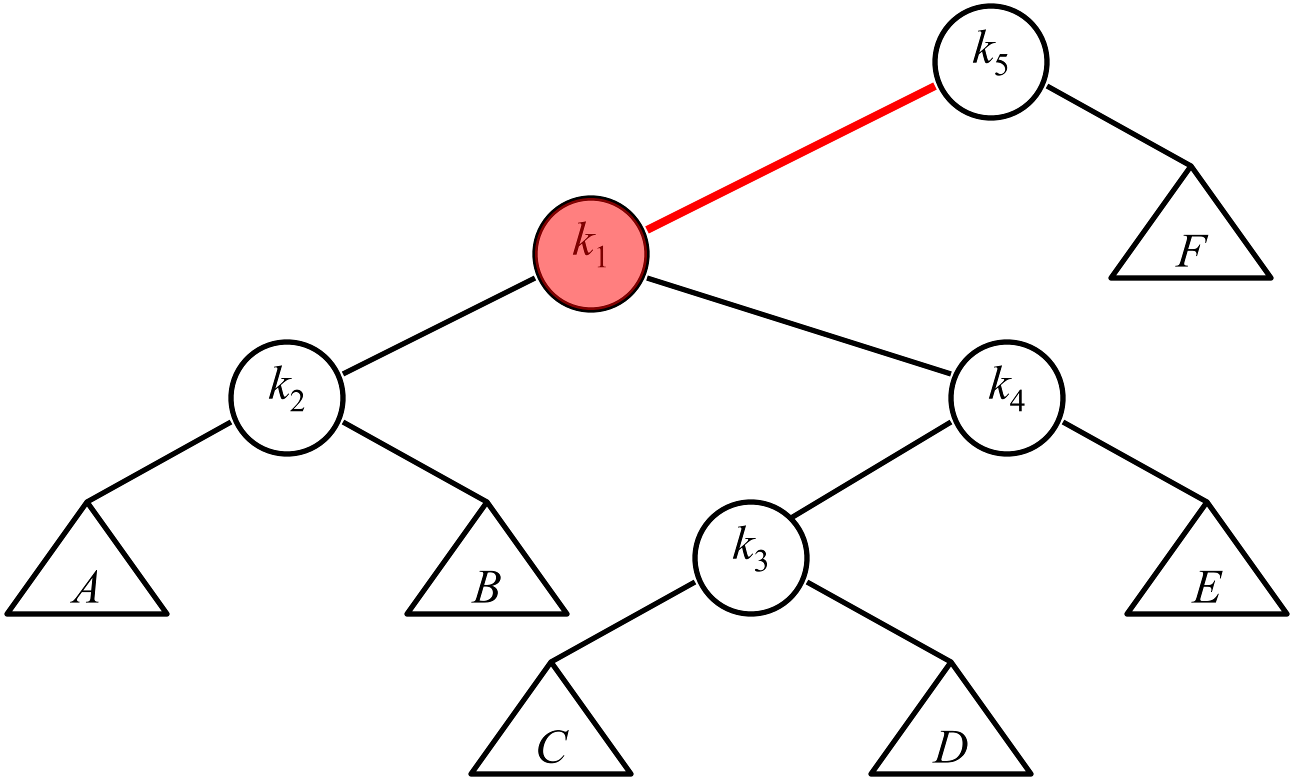

k_5
k_4
k_3
k_2
k_1
A
B
C
D
E
F

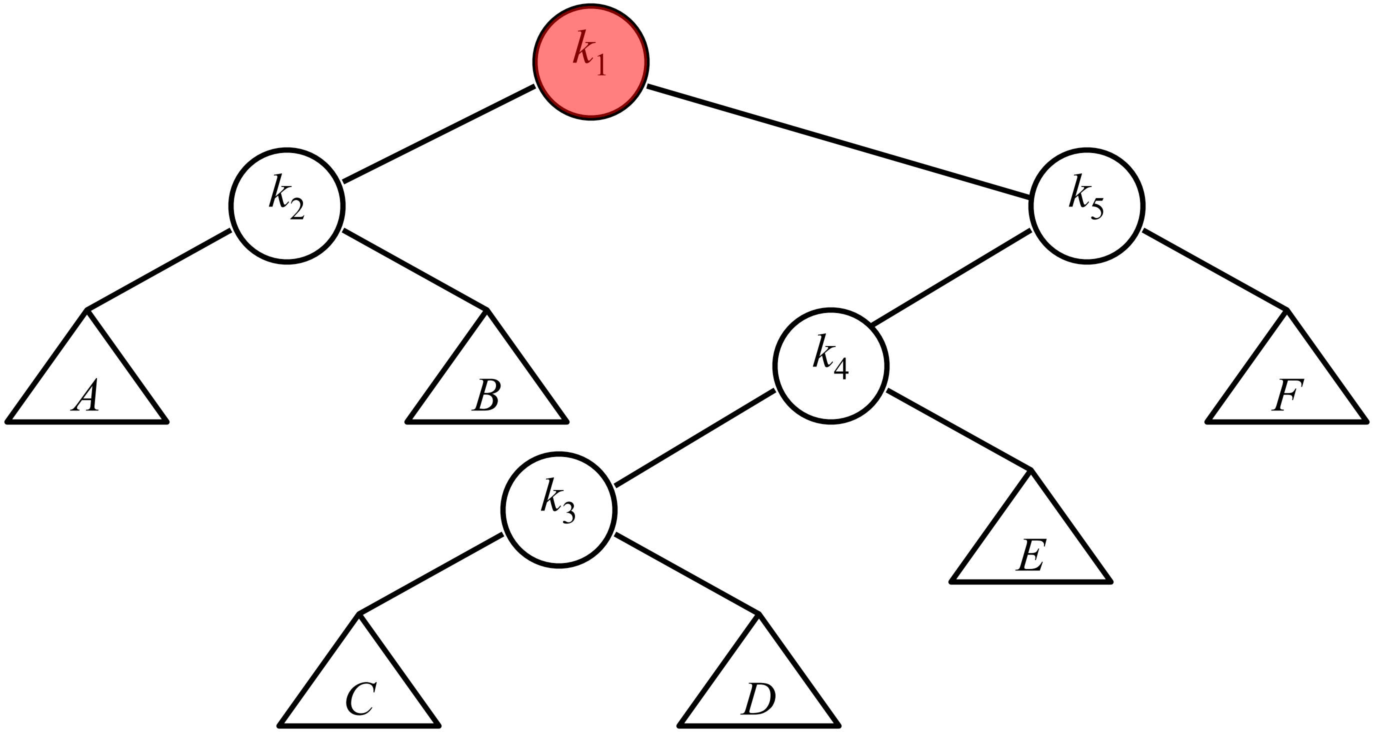

k_5
k_4
F
k_3
E
k_2
D
k_1
A
B
C

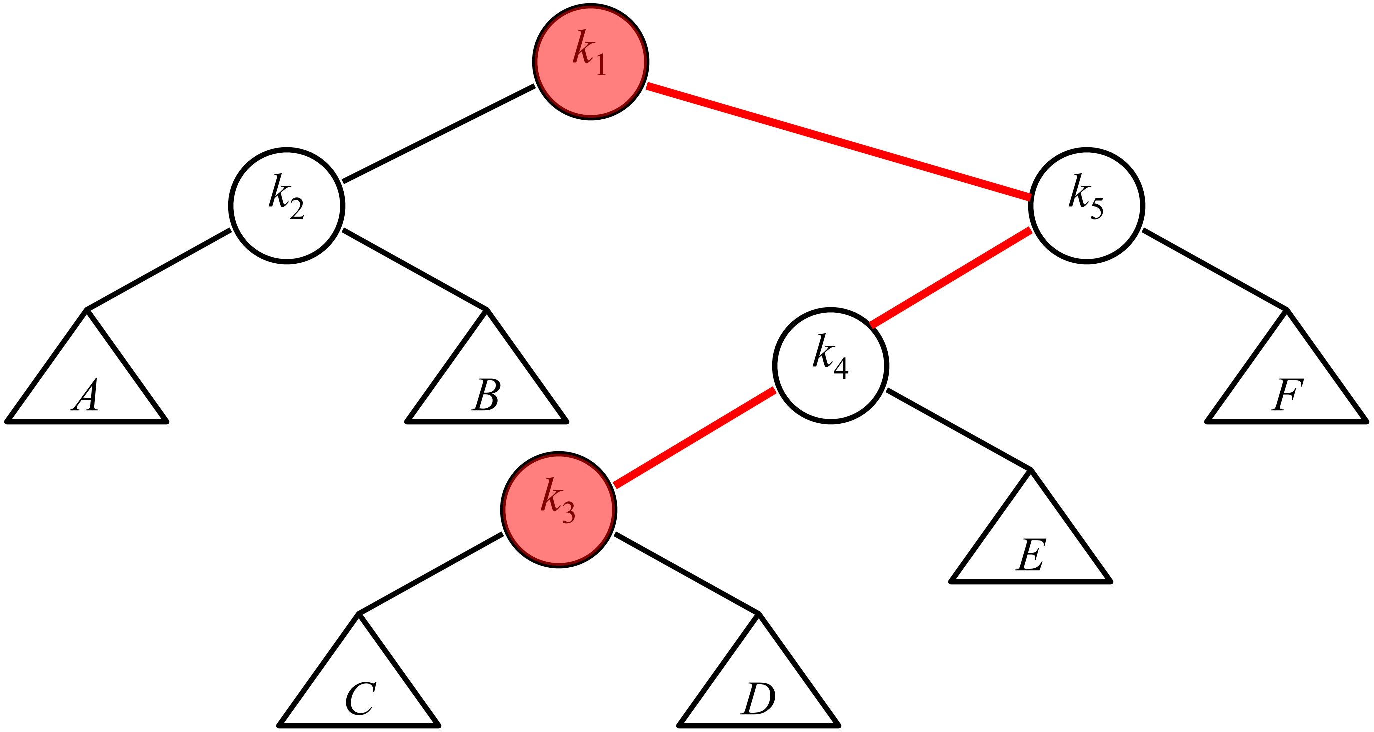

k_5
k_4
k_3
k_2
k_1
A
B
C
D
E
F

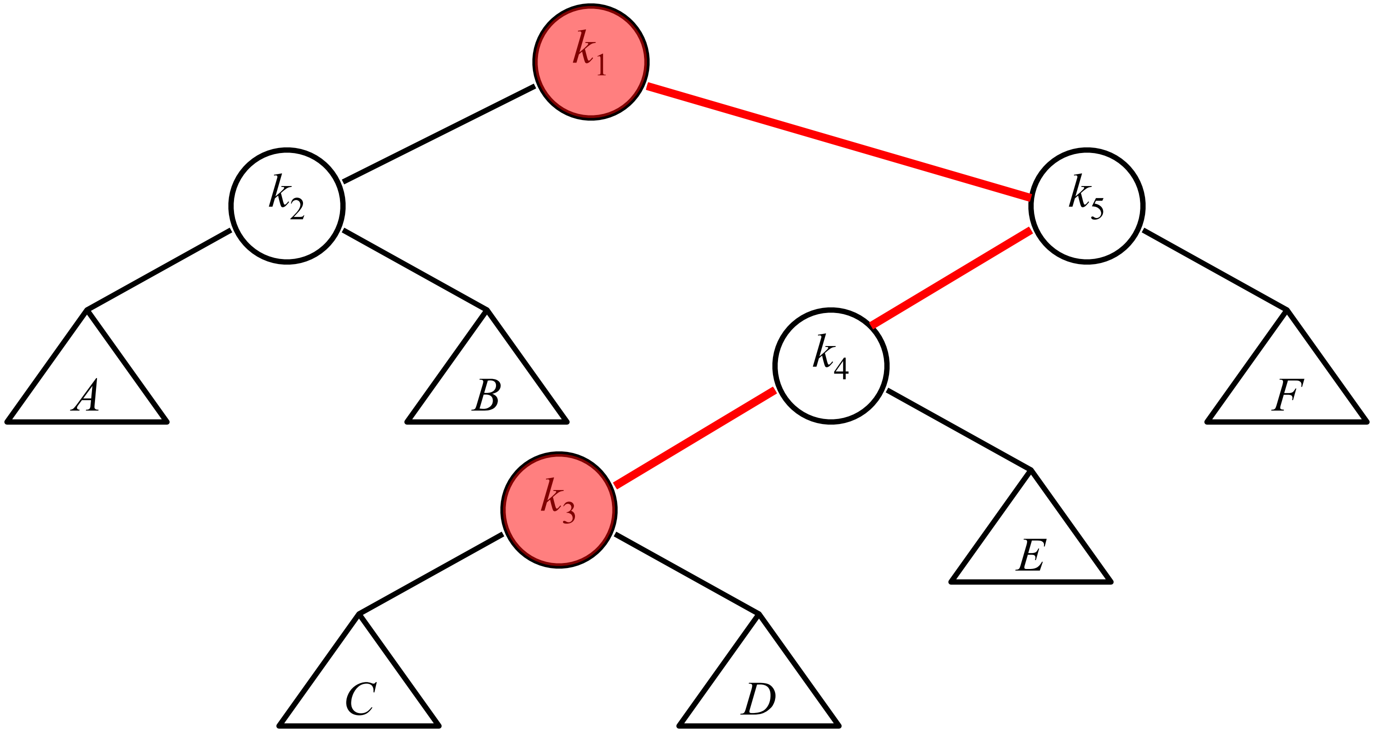

k_5
k_4
F
k_3
E
k_1
D
k_2
C
A
B

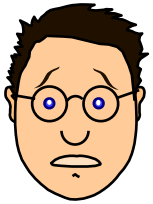

k_5
k_4
k_1
k_2
k_3
A
B
C
D
E
F


k_5
k_1
F
k_2
k_4
k_3
A
B
E
C
D


k_1
k_2
k_5
k_4
k_3
A
B
C
D
E
F


k_1
k_2
k_5
k_4
k_3
A
B
F
E
C
D



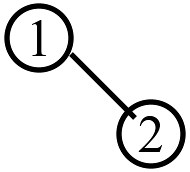


Does NOT work!

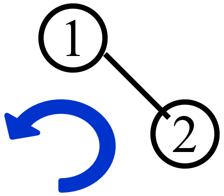
The rotation pushes other nodes deeper

An even worse case:

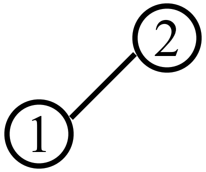
An even worse case:



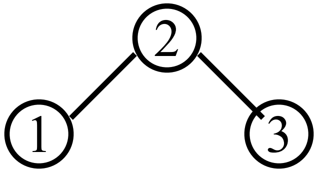
An even worse case:



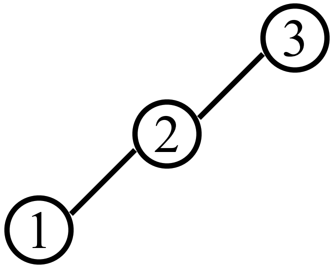
An even worse case:



An even worse case:

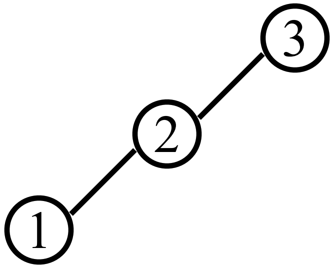


An even worse case:



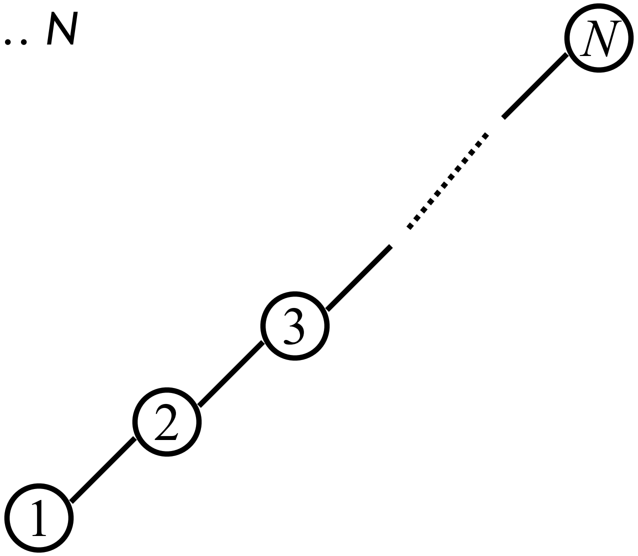
An even worse case:

Insert: 1, 2, 3, … *N*



An even worse case:

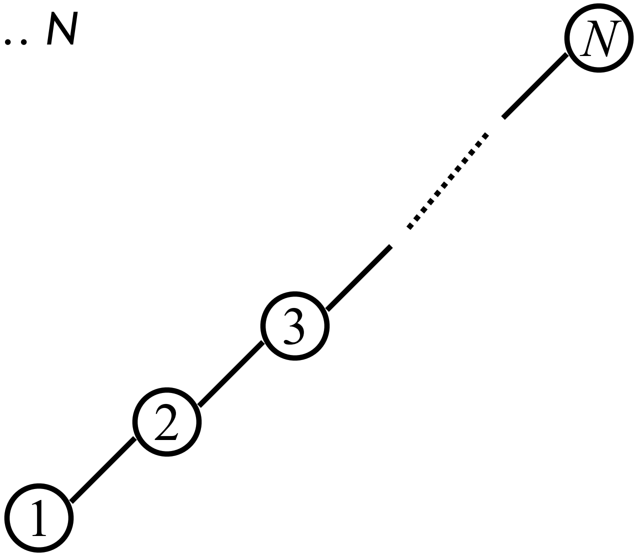
Insert: 1, 2, 3, … *N*



An even worse case:

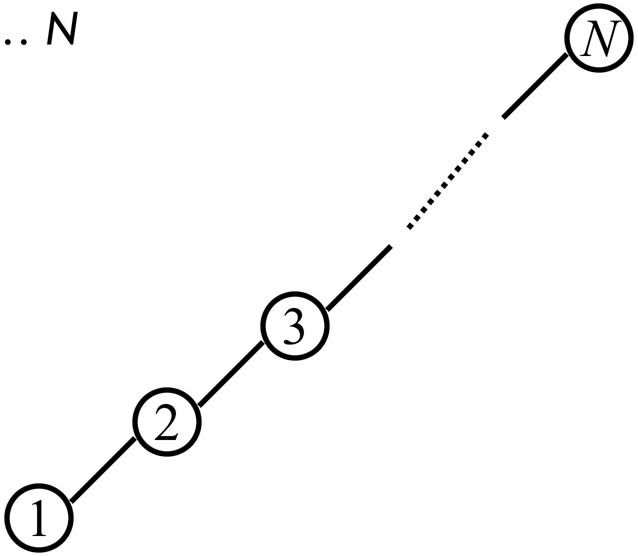
Insert: 1, 2, 3, … *N*

Find: 1

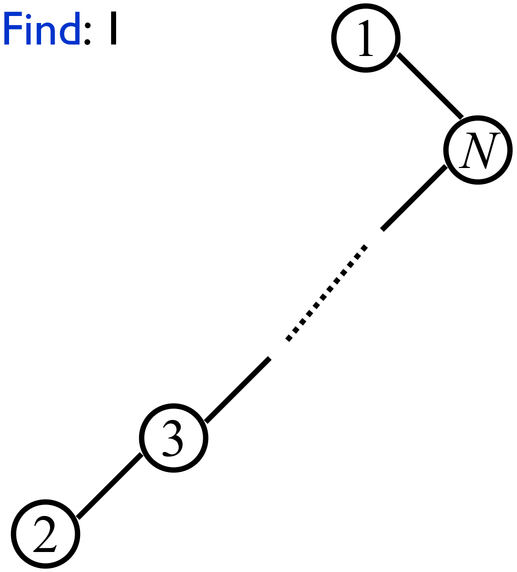


An even worse case:

Insert: 1, 2, 3, … *N*

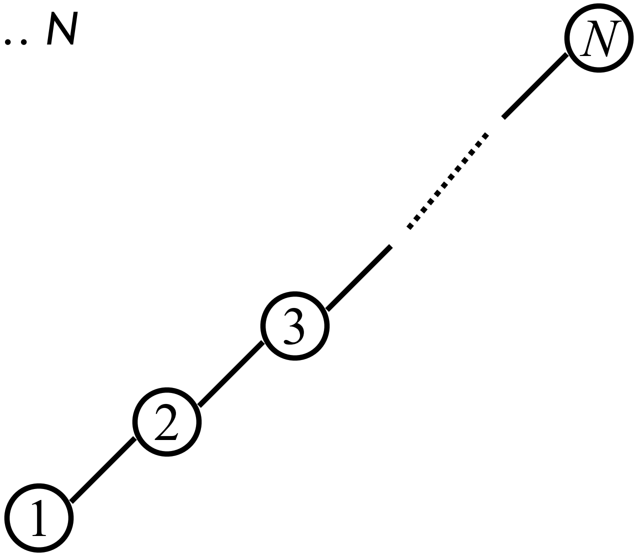


Find: 1

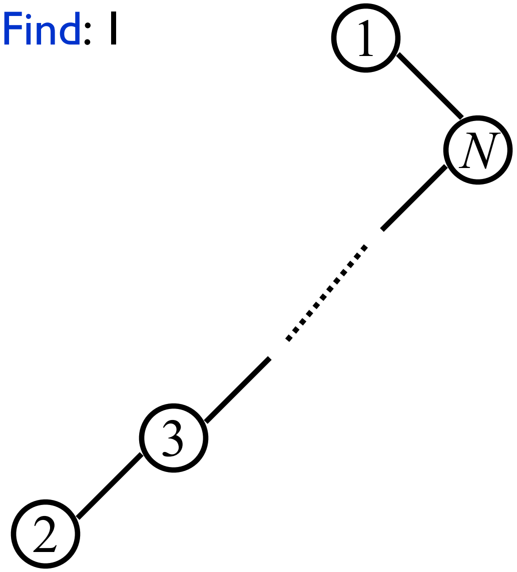


An even worse case:

Insert: 1, 2, 3, … *N*



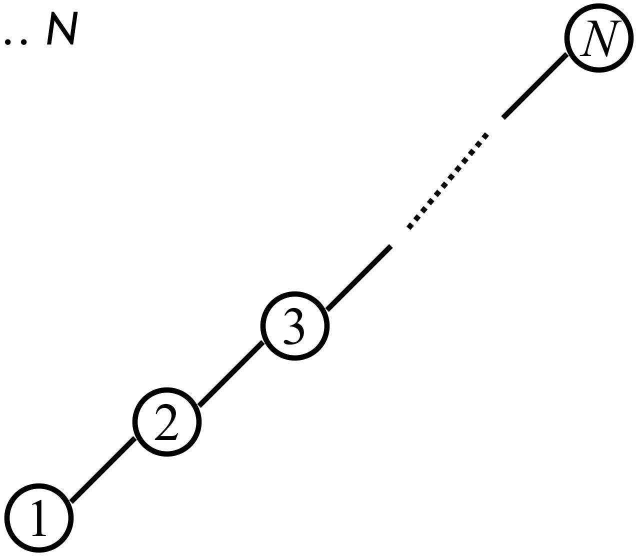
Find: 1



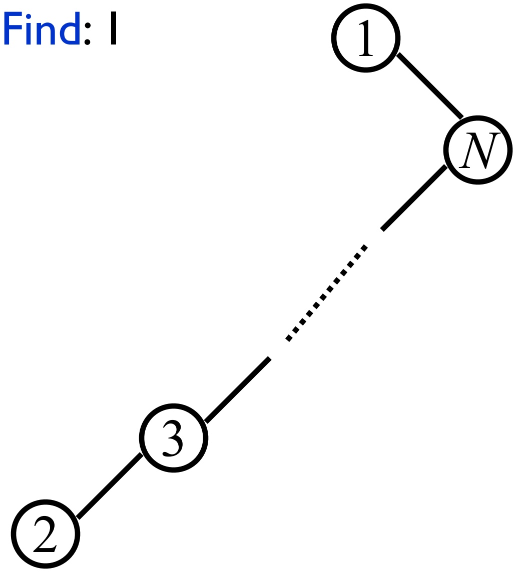
Find: 2

An even worse case:

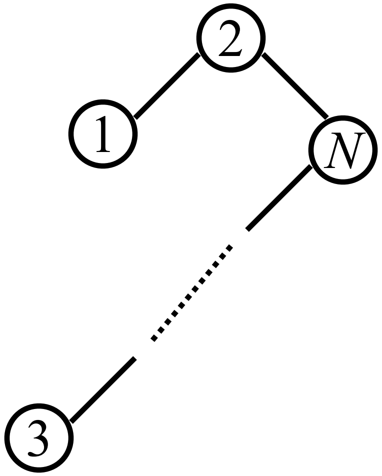
Insert: 1, 2, 3, … *N*



Find: 1

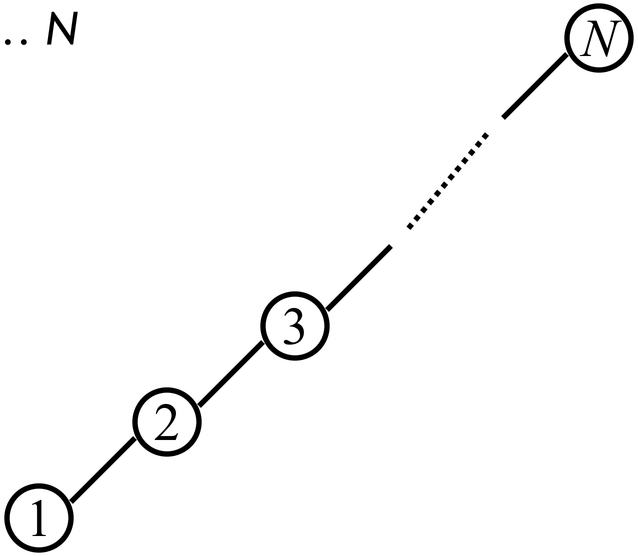


Find: 2

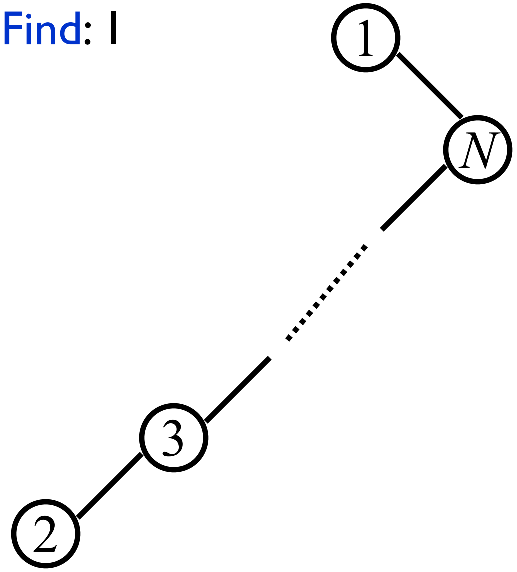


An even worse case:

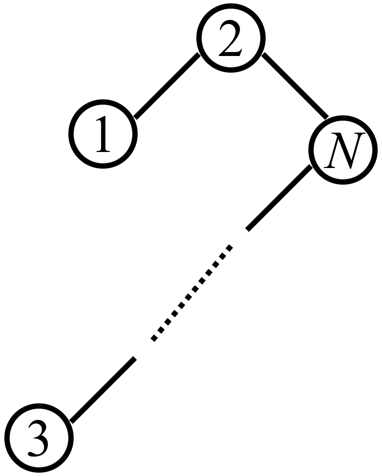
Insert: 1, 2, 3, … *N*



Find: 1



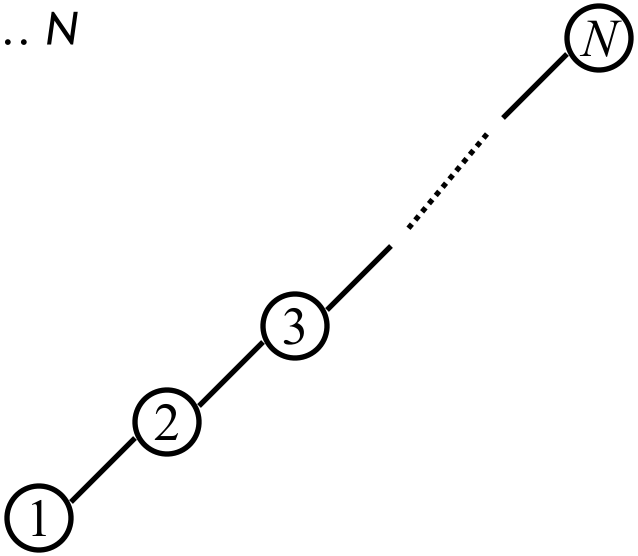
Find: 2



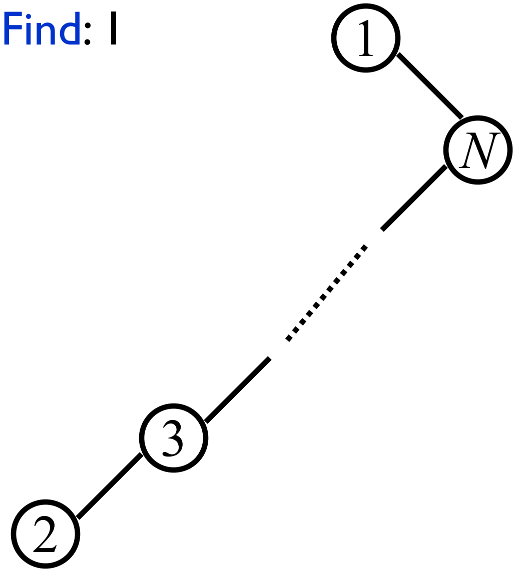
…… Find: *N*

An even worse case:

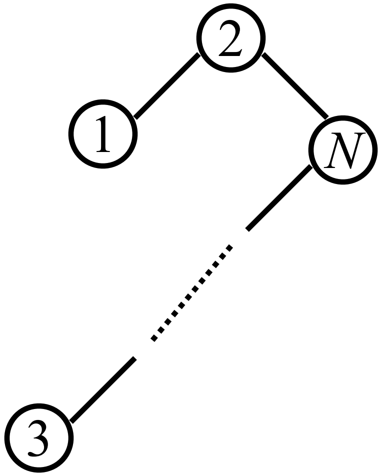
Insert: 1, 2, 3, … *N*



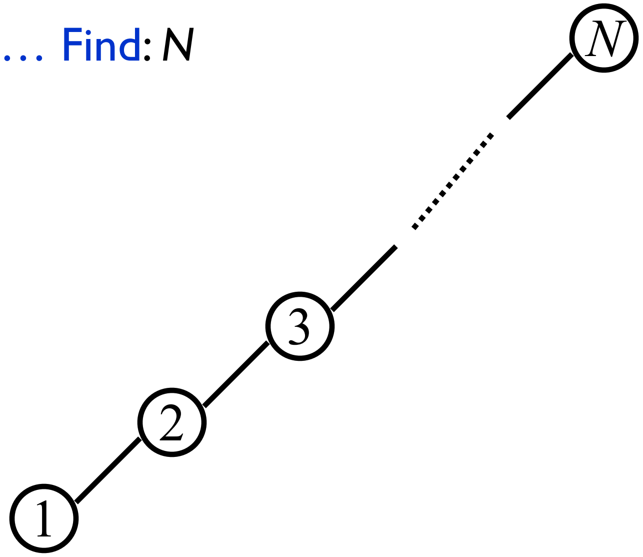
Find: 1



Find: 2

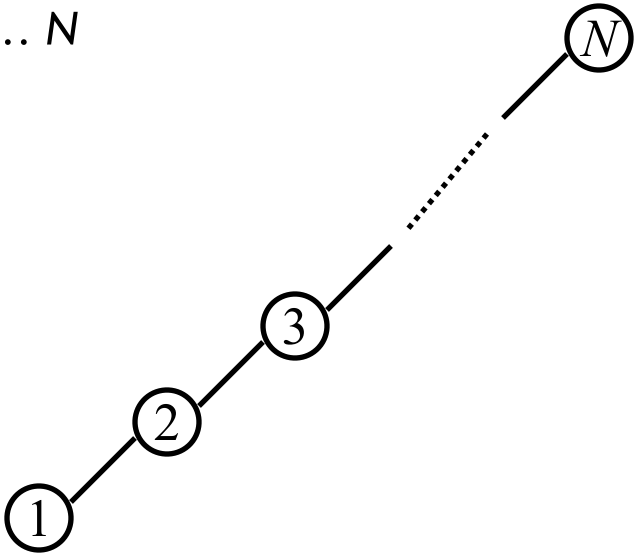


…… Find: *N*

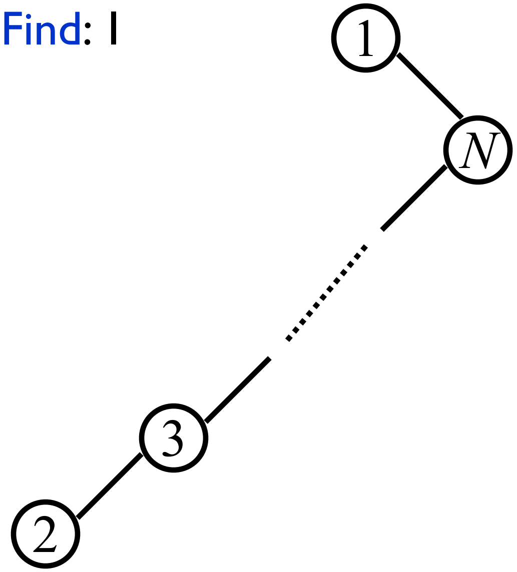


An even worse case:

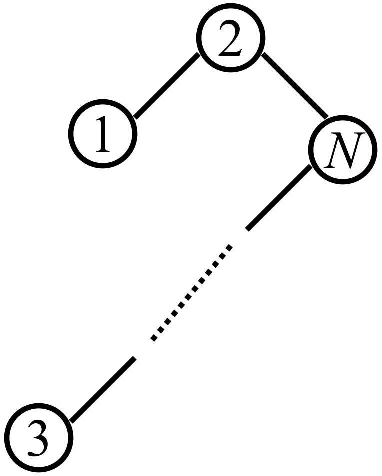
Insert: 1, 2, 3, … *N*



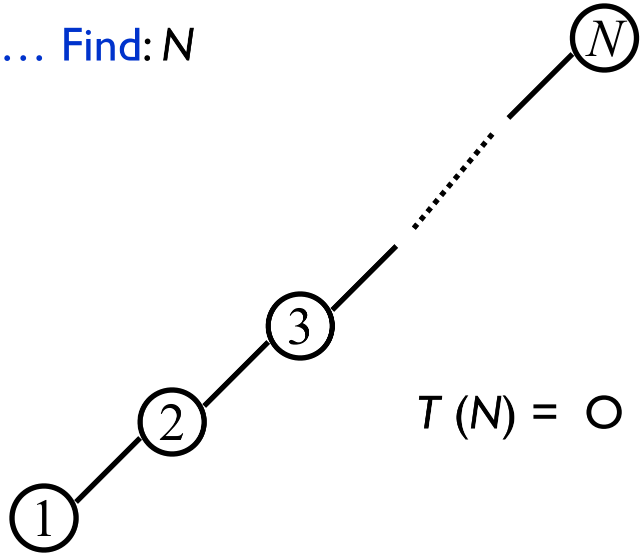
Find: 1



Find: 2



…… Find: *N*



$T(N) = O(N^2)$

Try again -- For any nonroot node *X* , denote its parent by *P* and grandparent by *G* :

Try again -- For any nonroot node *X* , denote its parent by *P* and grandparent by *G* :

Zig Case 1: *P* is the root

Try again -- For any nonroot node *X* , denote its parent by *P* and grandparent by *G* :

Zig Case 1: *P* is the root → Rotate *X* and *P*

Try again -- For any nonroot node *X* , denote its parent by *P* and grandparent by *G* :

Zig Case 1: *P* is the root → Rotate *X* and *P*

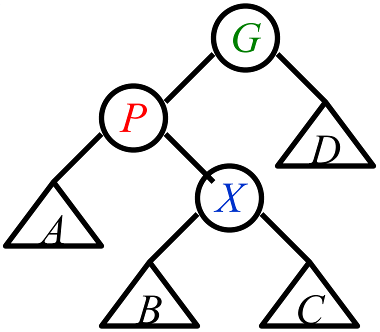
Case 2: *P* is not the root

Try again -- For any nonroot node *X* , denote its parent by *P* and grandparent by *G* :

Zig Case 1: *P* is the root ⟶ Rotate *X* and *P*

Case 2: *P* is not the root

Zig-zag

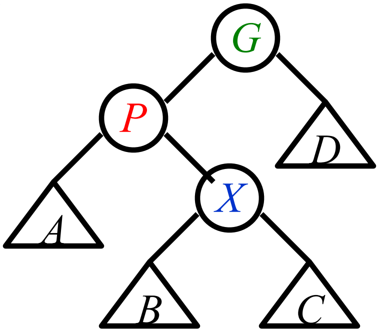


Try again -- For any nonroot node *X* , denote its parent by *P* and grandparent by *G* :

Zig Case 1: *P* is the root → Rotate *X* and *P*

Case 2: *P* is not the root

Zig-zag



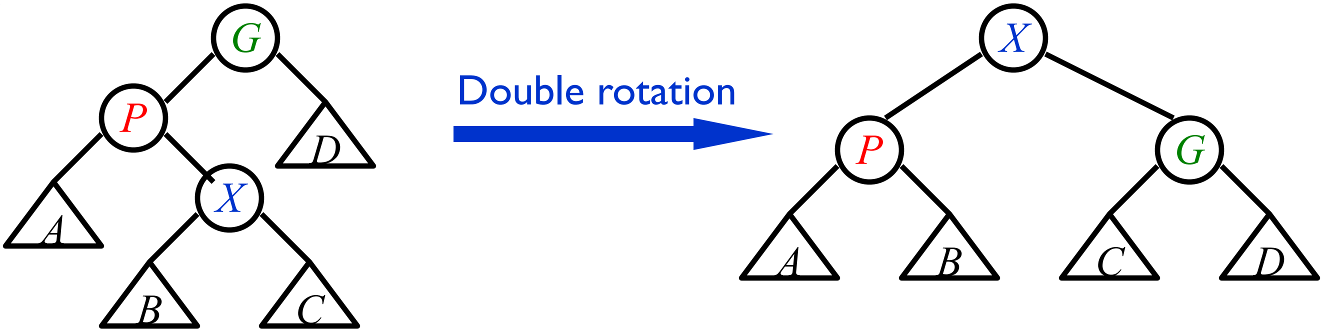
Double rotation →

Try again -- For any nonroot node *X* , denote its parent by *P* and grandparent by *G* :

Zig Case 1: *P* is the root → Rotate *X* and *P*

Case 2: *P* is not the root

Zig-zag

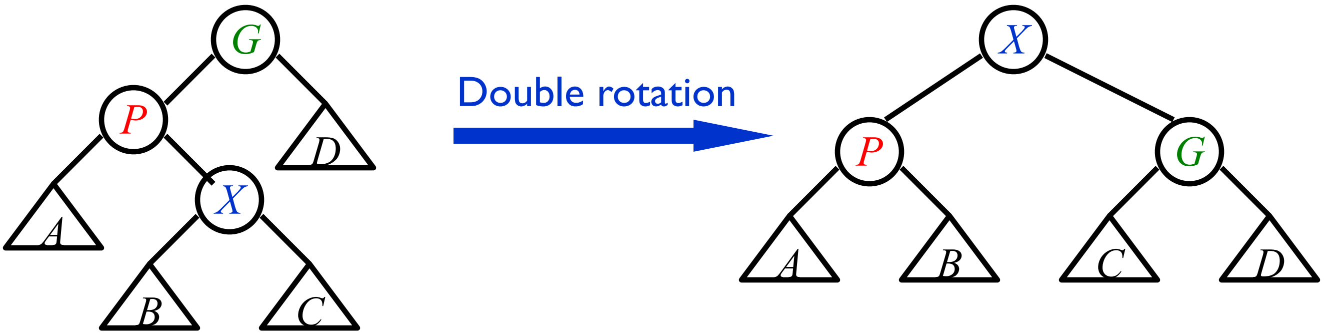


Try again -- For any nonroot node *X* , denote its parent by *P* and grandparent by *G* :

Zig Case 1: *P* is the root → Rotate *X* and *P*

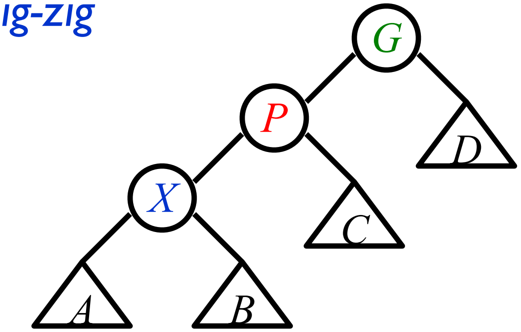
Case 2: *P* is not the root

Zig-zag



Double rotation

Zig-zig

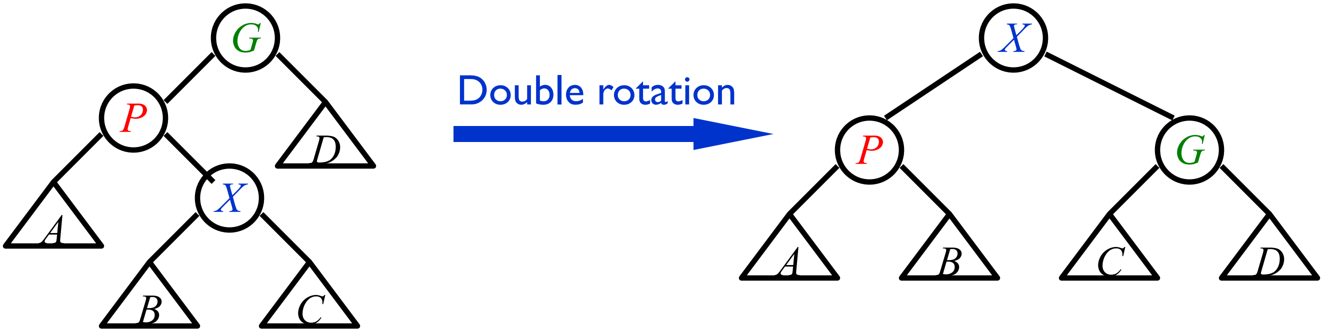


Try again -- For any nonroot node *X* , denote its parent by *P* and grandparent by *G* :

Zig Case 1: *P* is the root → Rotate *X* and *P*

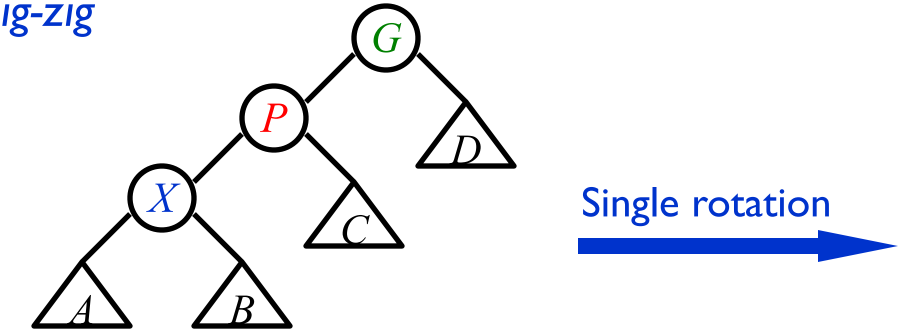
Case 2: *P* is not the root

Zig-zag



Double rotation

Zig-zig



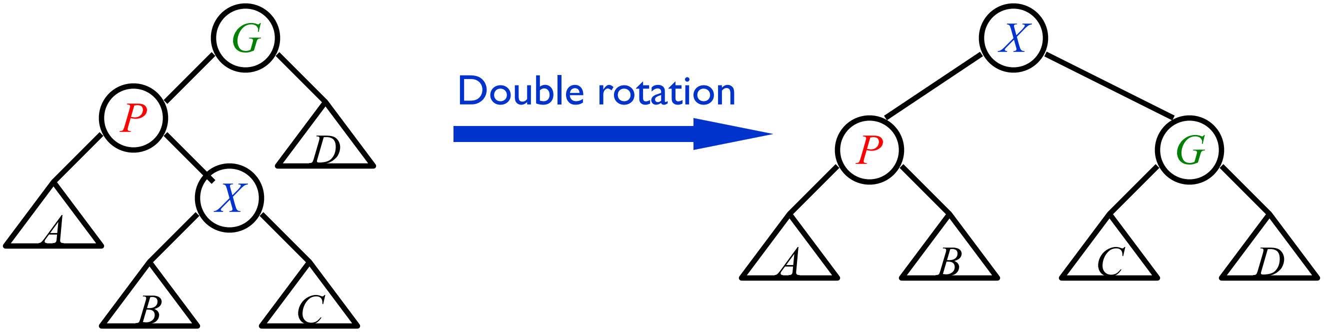
Single rotation

Try again -- For any nonroot node *X* , denote its parent by *P* and grandparent by *G* :

Zig Case 1: *P* is the root → Rotate *X* and *P*

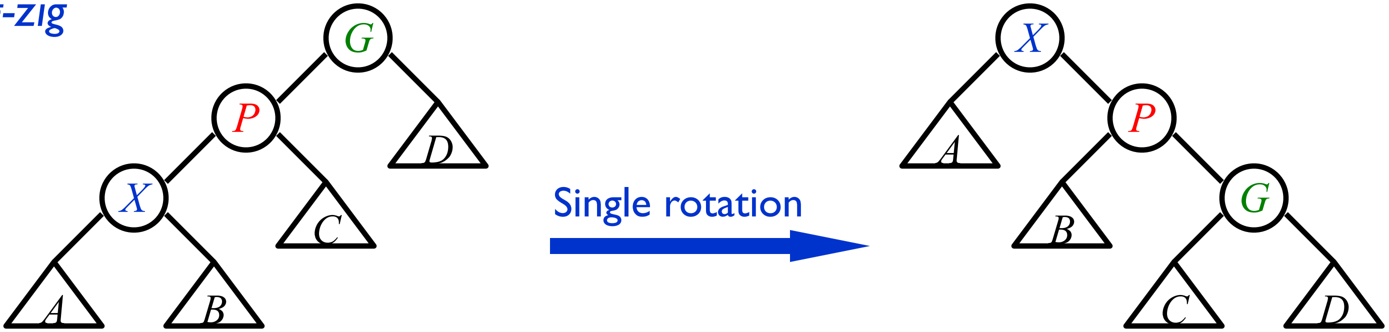
Case 2: *P* is not the root

Zig-zag



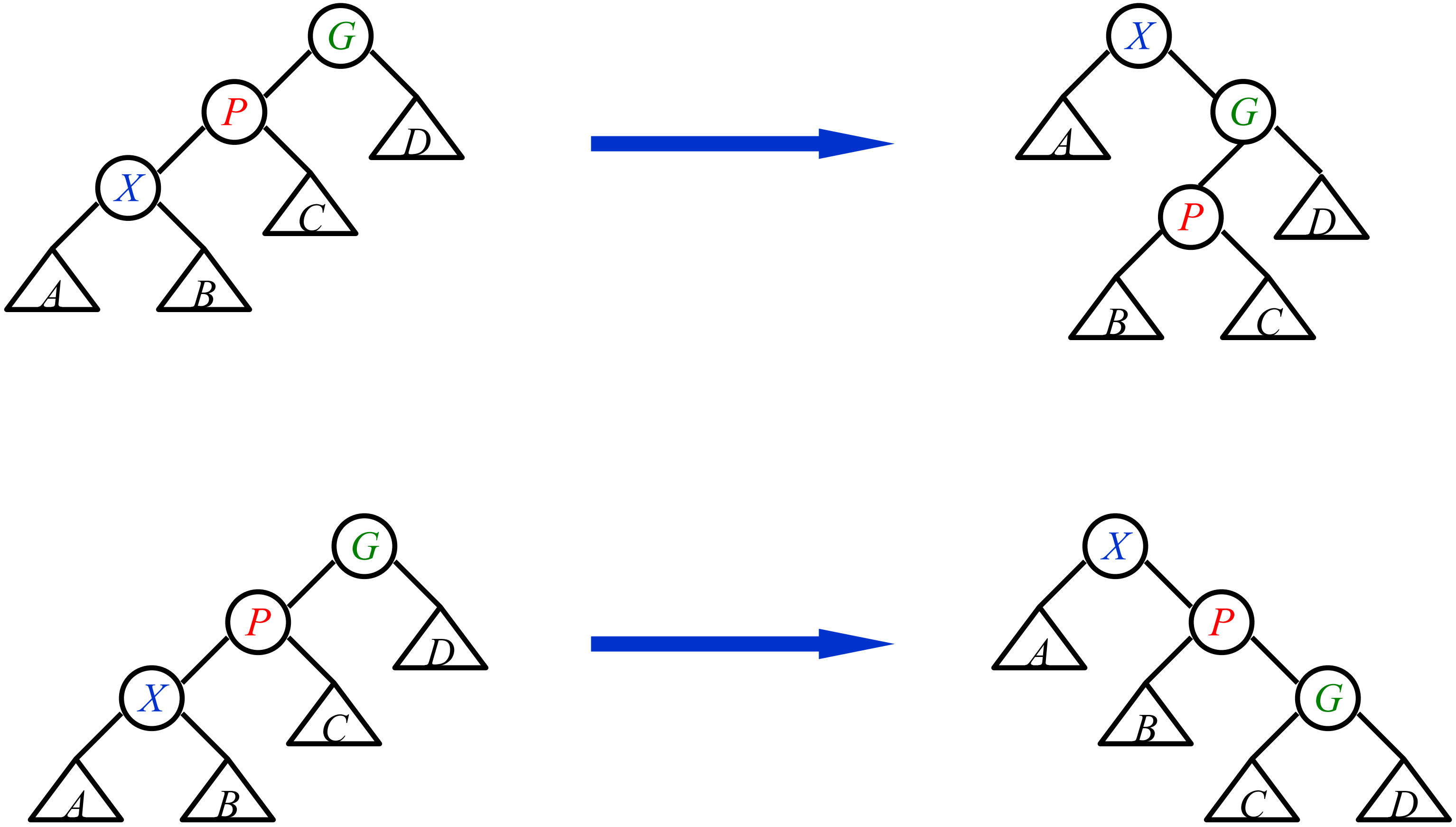
Double rotation

Zig-zig

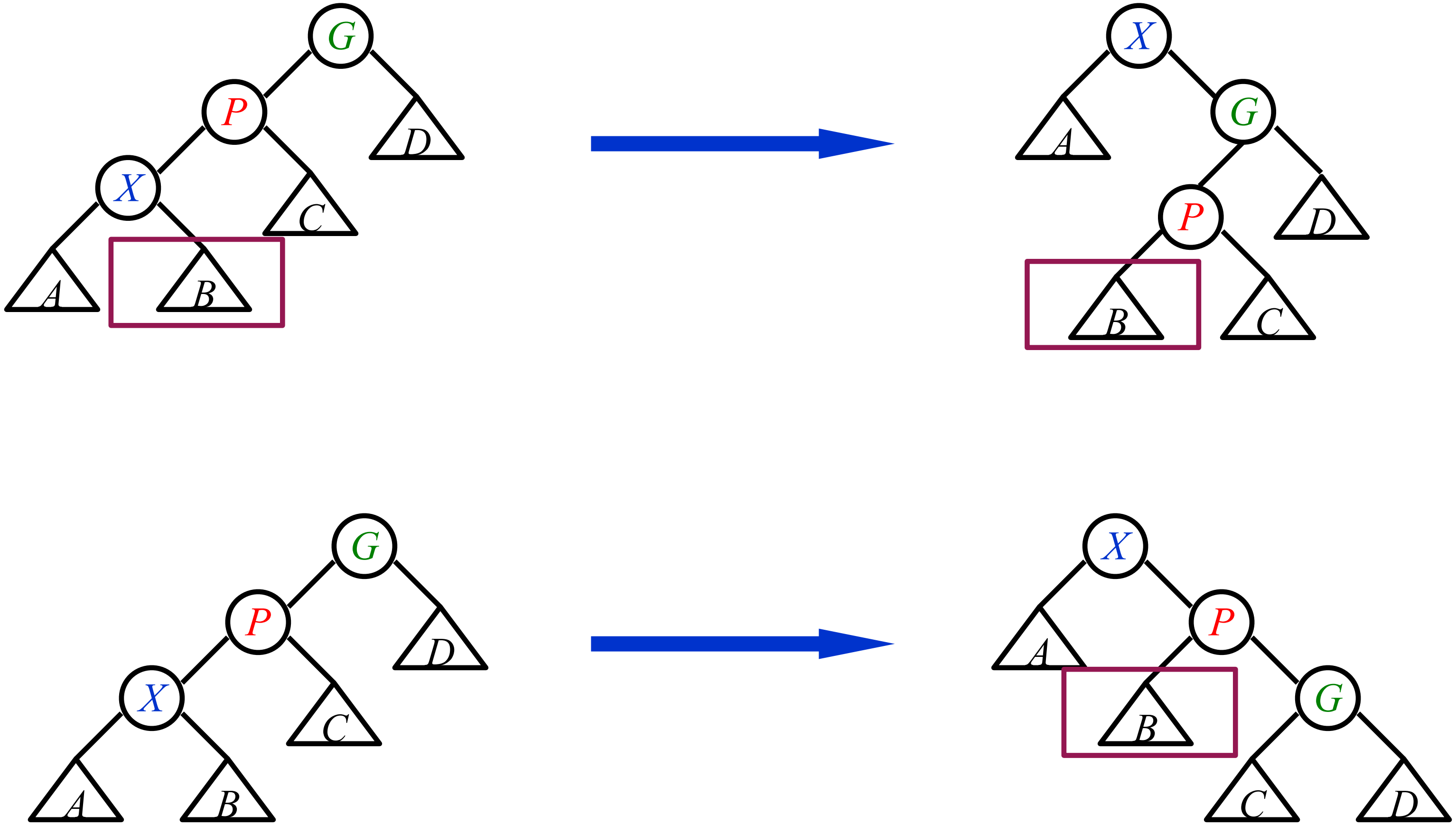


Single rotation

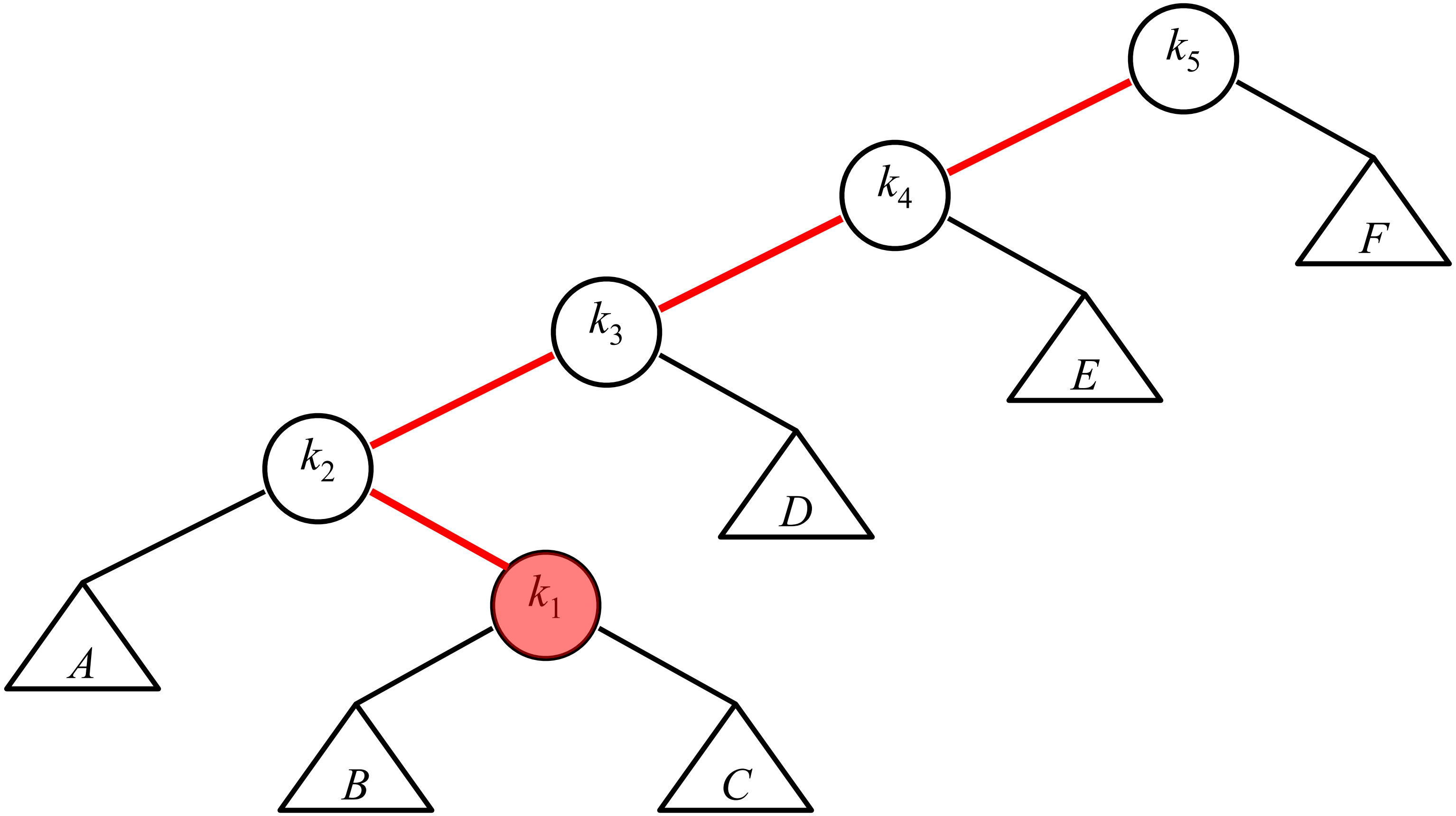
Compare the Zig-zig case:

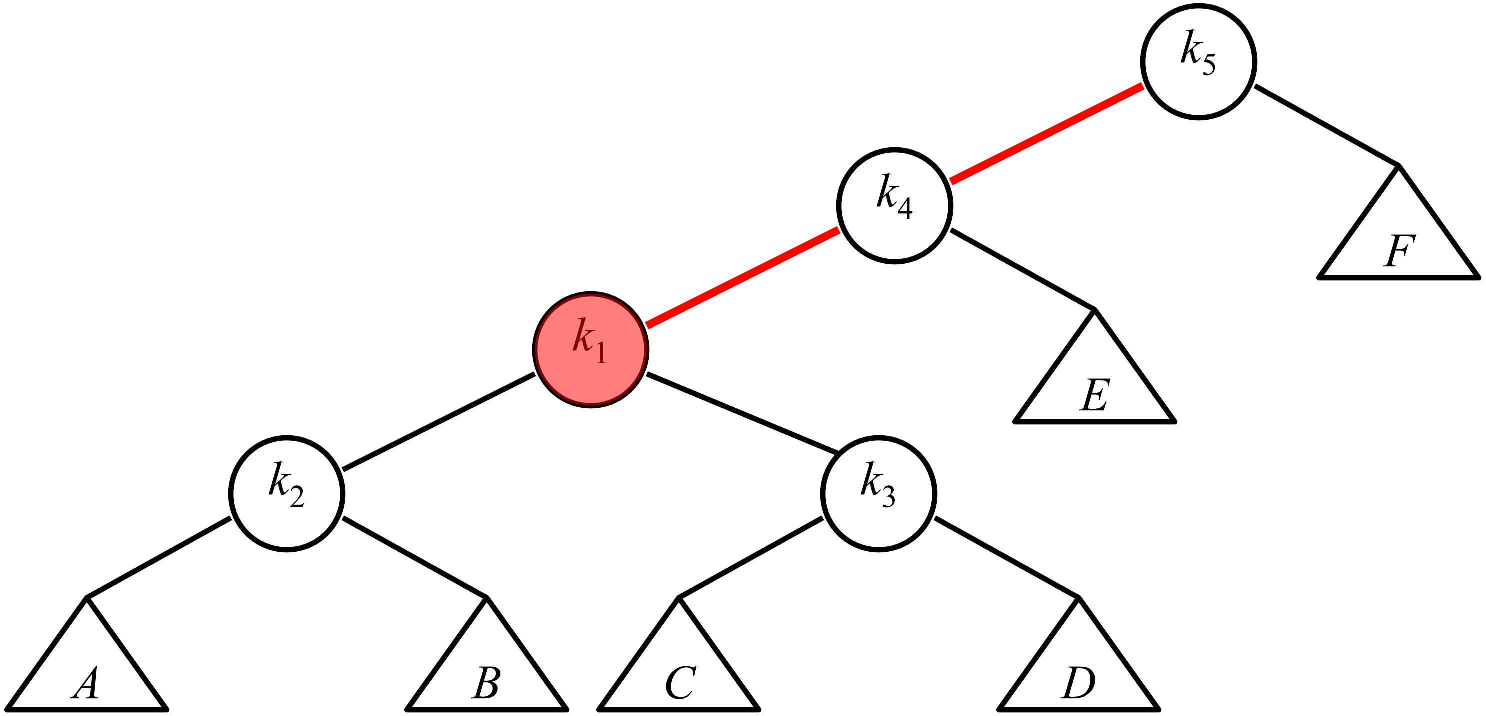


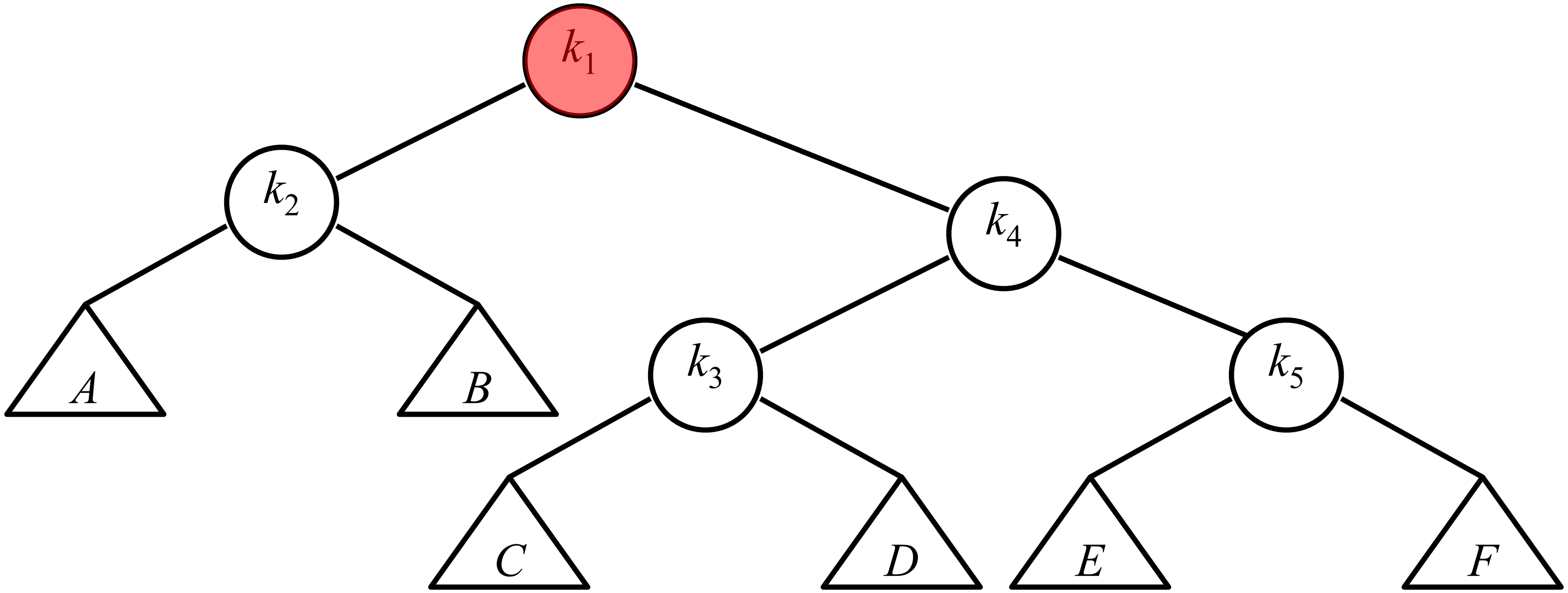
Compare the Zig-zig case:



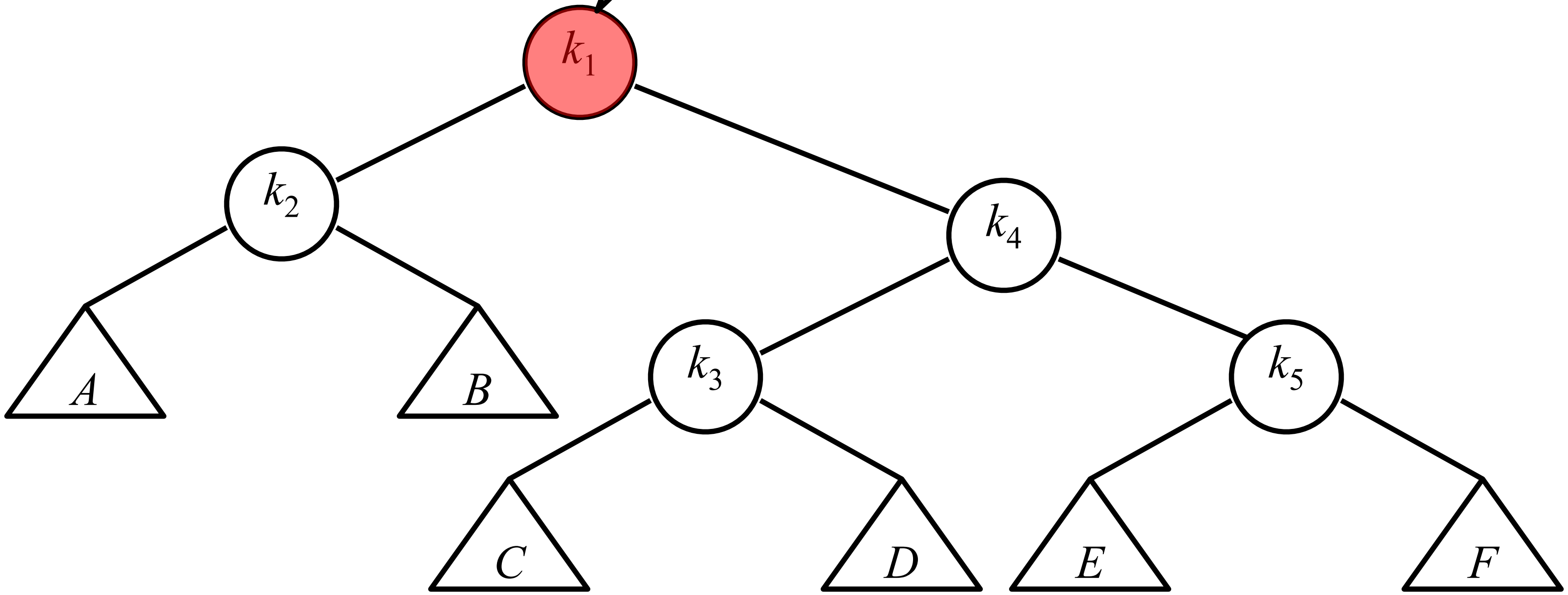
For zig-zig case, the right child of the node on splaying always goes deep.
The key is to make it go slower.


k_5
k_4
k_3
k_2
k_1
A
B
C
D
E
F


k_5
k_4
k_1
k_2
k_3
A
B
C
D
E
F

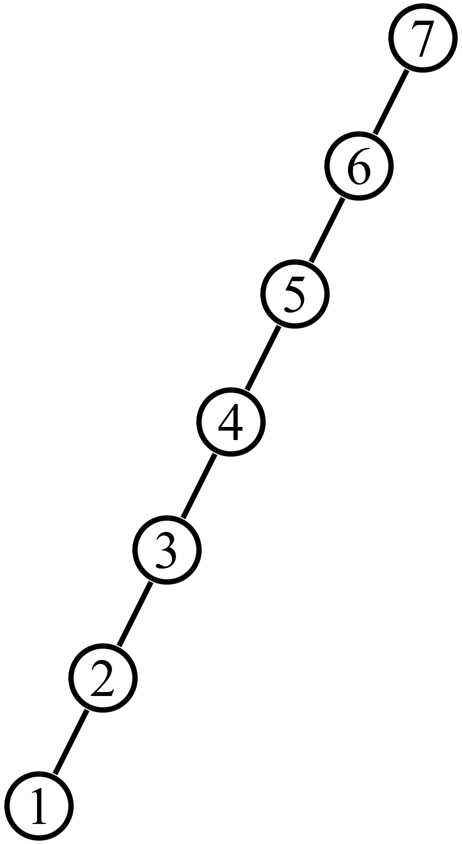

k_1
k_2
k_4
k_3
k_5
A
B
C
D
E
F

Splaying not only moves the accessed node to the root, but also roughly halves the depth of most nodes on the path.

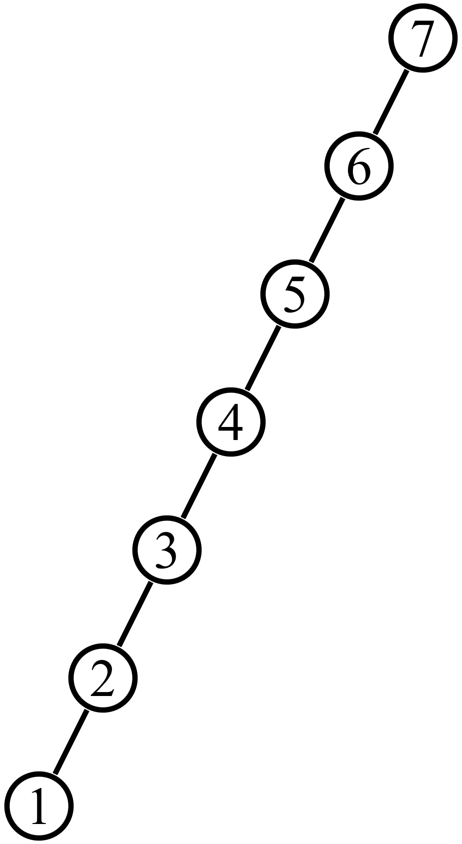


Insert: 1, 2, 3, 4, 5, 6, 7

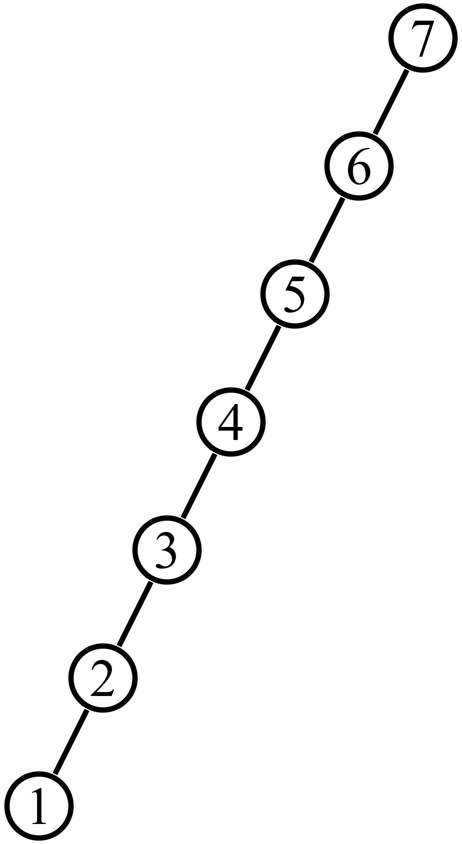
Insert: 1, 2, 3, 4, 5, 6, 7



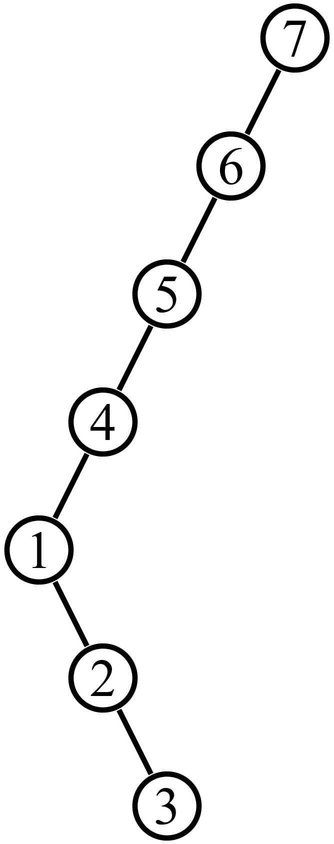
Insert: 1, 2, 3, 4, 5, 6, 7 Find: 1



Insert: 1, 2, 3, 4, 5, 6, 7

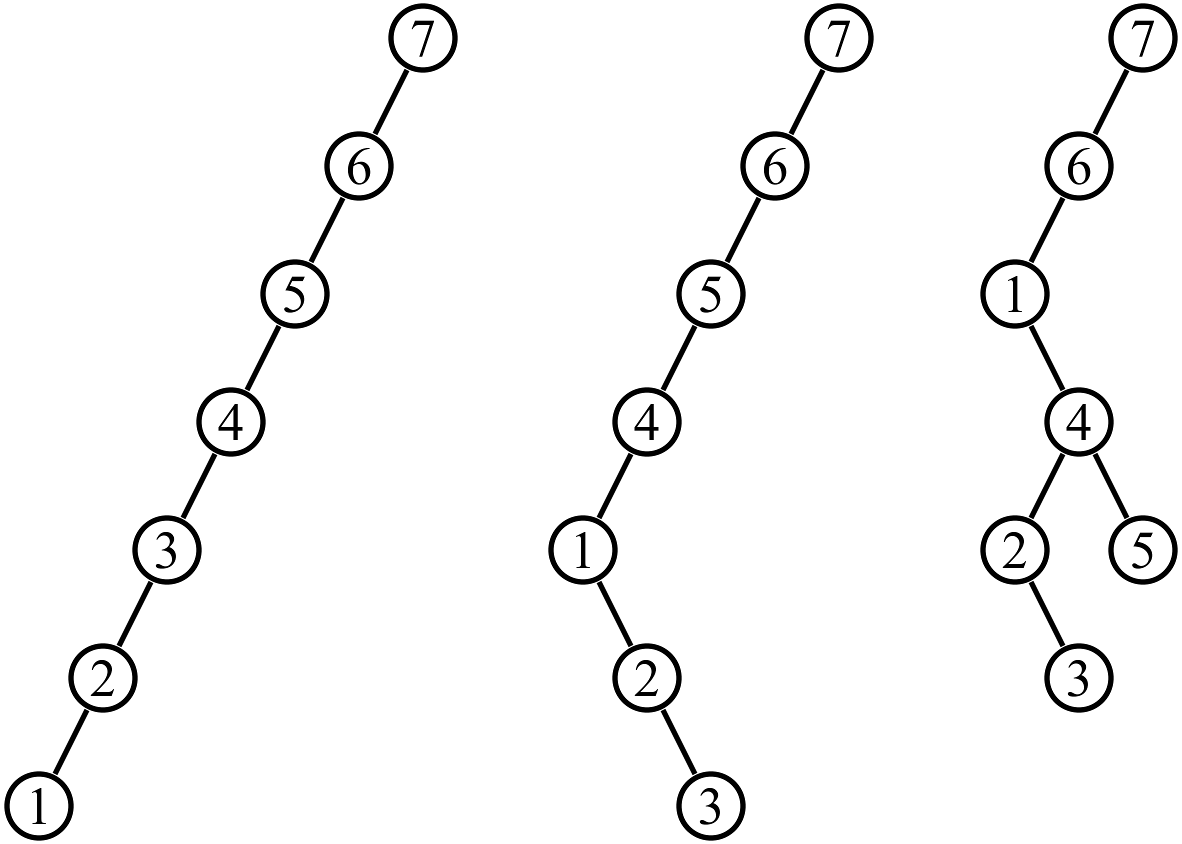


Find: 1

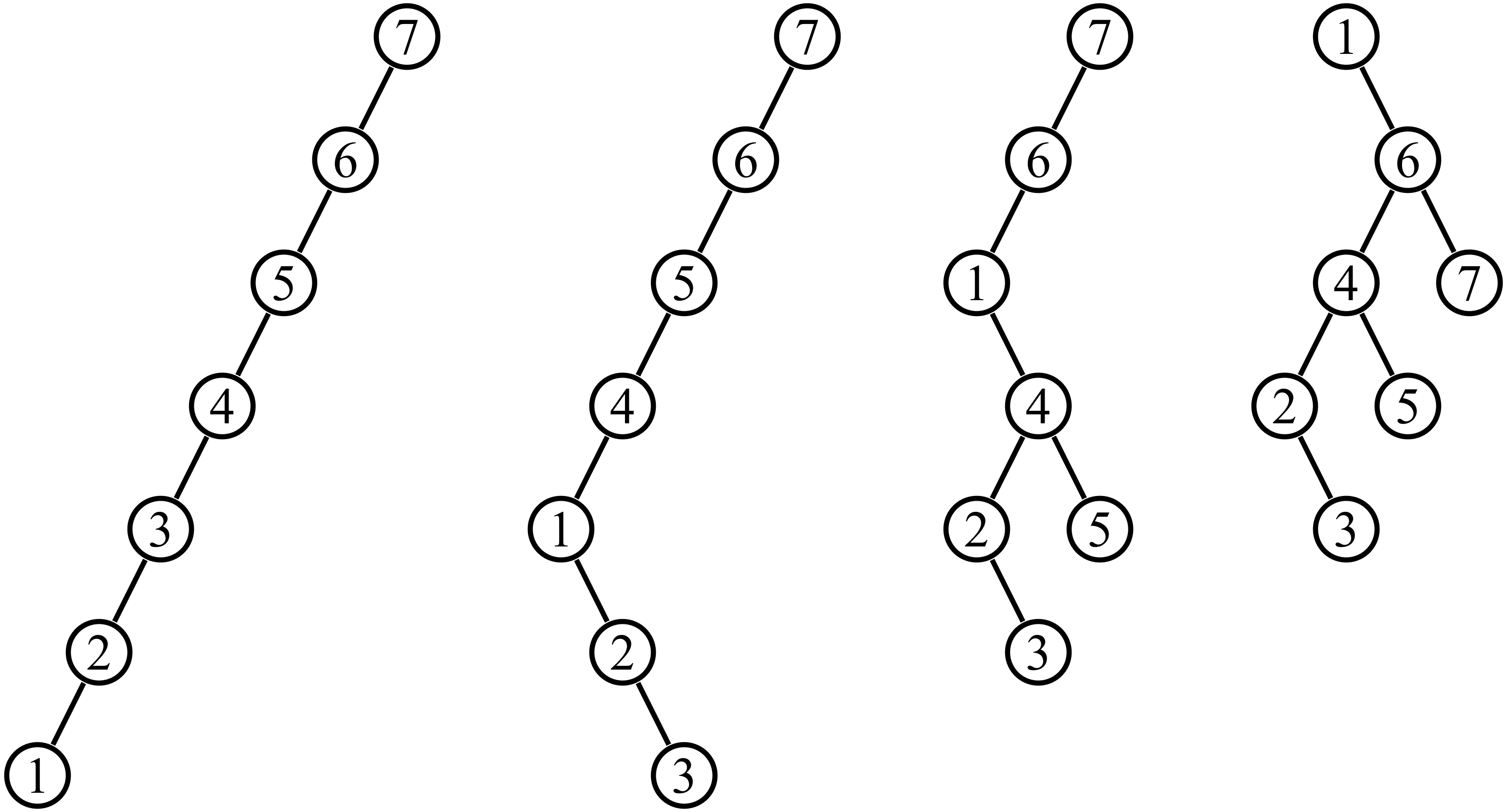


Insert: 1, 2, 3, 4, 5, 6, 7

Find: 1

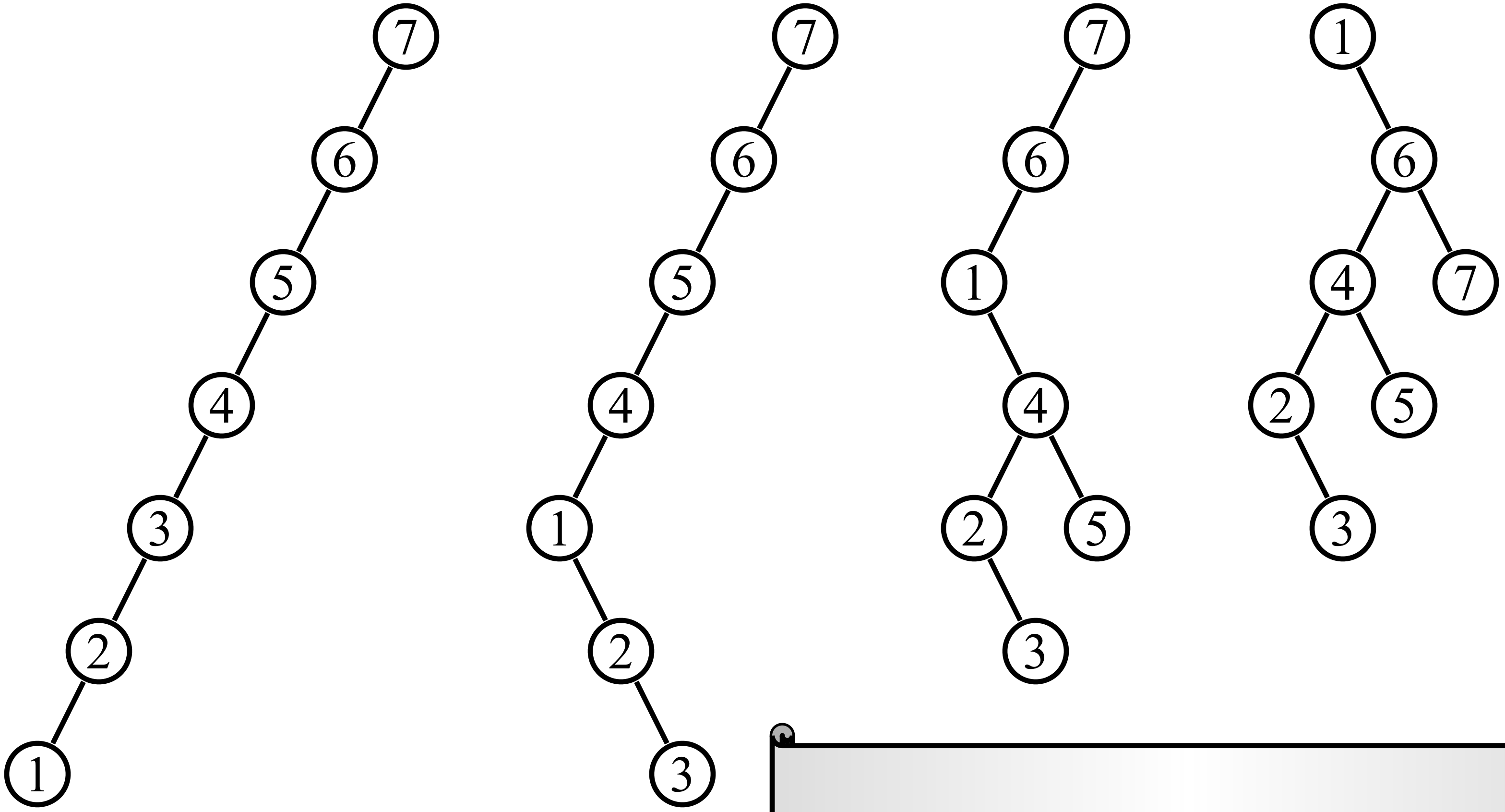


Insert: 1, 2, 3, 4, 5, 6, 7   Find: 1



Insert: 1, 2, 3, 4, 5, 6, 7

Find: 1



Read the 32-node example given in [Weiss] Figures 4.52 – 4.60

Operations on Splay Trees

Operations on Splay Trees

Deletions:

Operations on Splay Trees

Deletions:

☞ Step 1: Find *X* ;

X will be at the root due to splaying.

Operations on Splay Trees

Deletions:

☞ Step 1: Find *X* ;

X will be at the root due to splaying.

☞ Step 2: Remove *X* ;

Operations on Splay Trees

Deletions:

☞ Step 1: Find *X* ;

X will be at the root due to splaying.

☞ Step 2: Remove *X* ;

There will be two subtrees T_L and T_R .

Operations on Splay Trees

Deletions:

☞ Step 1: Find X ;

X will be at the root due to splaying.

☞ Step 2: Remove X ;

There will be two subtrees T_L and T_R .

☞ Step 3: FindMax (T_L) ;

Operations on Splay Trees

Deletions:

☞ Step 1: Find X ;

X will be at the root due to splaying.

☞ Step 2: Remove X ;

There will be two subtrees T_L and T_R .

☞ Step 3: FindMax (T_L) ;

The largest element will be the root of T_L , and *has no right child.*

Operations on Splay Trees

Deletions:

☞ Step 1: Find X ;

X will be at the root due to splaying.

☞ Step 2: Remove X ;

There will be two subtrees T_L and T_R .

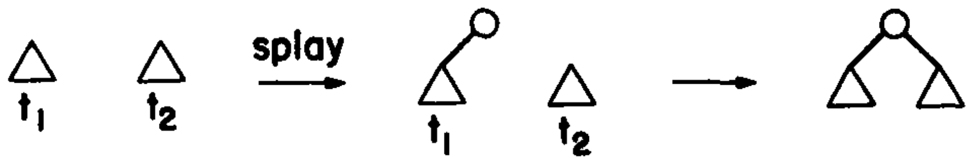
☞ Step 3: FindMax (T_L) ;

The largest element will be the root of T_L , and *has no right child.*

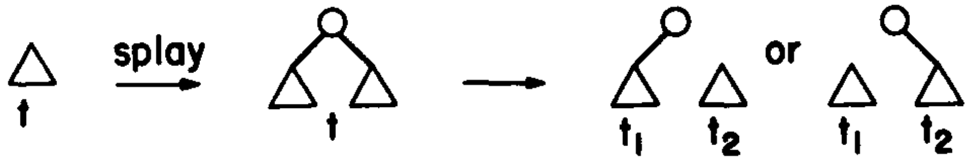
☞ Step 4: Make T_R the right child of the root of T_L .

Operations on Splay Trees

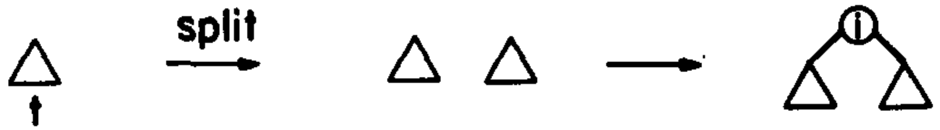
Join(t1, t2):



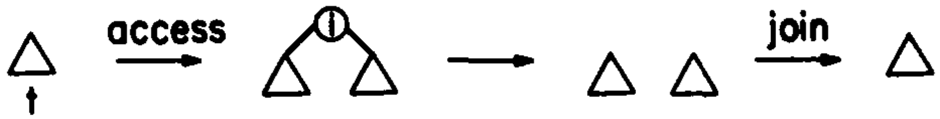
Split(i, t):



Insert(i, t):



Delete(i, t):



All operations involve a series of splay steps.
Check the details in the "Self-adjusting binary search trees" paper.
Next, we study the complexity of splay tree operations.

Outline:
Balanced Binary Search Trees (1)

- Binary search trees
- AVL trees
- Splay trees
- Amortized analysis
- Take-home messages

Amortized Analysis

Amortized Analysis

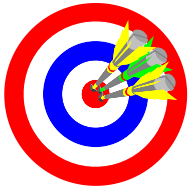
Target : Any M consecutive operations take at most $O(M \log N)$ time.

Amortized Analysis

Target : Any *M* consecutive operations take at most O(*M* log *N*) time.

-- *Amortized* time bound

Amortized Analysis

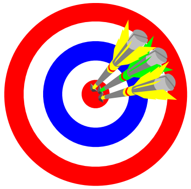


Target : Any *M* consecutive operations take at most O(*M* log *N*) time.

-- *Amortized* time bound

worst-case bound   amortized bound   average-case bound

Amortized Analysis

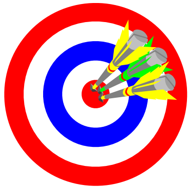


Target : Any *M* consecutive operations take at most O(*M* log *N*) time.

-- *Amortized* time bound

worst-case bound ≥ amortized bound average-case bound

Amortized Analysis



Target : Any *M* consecutive operations take at most O(*M* log *N*) time.

-- *Amortized* time bound

worst-case bound ≥ amortized bound ≥ average-case bound

Amortized Analysis

Target : Any M consecutive operations take at most O(M log N) time.

-- *Amortized* time bound

worst-case bound ≥ amortized bound ≥ average-case bound

Probability
is *not* involved

Amortized Analysis

Target : Any M consecutive operations take at most O(M log N) time.

-- *Amortized* time bound

worst-case bound ≥ amortized bound ≥ average-case bound

Probability
is *not* involved

☞ Aggregate analysis

☞ Accounting method

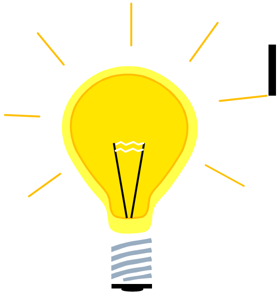
☞ Potential method

Aggregate Method

Aggregate Method

Idea : Show that for all *n*, a sequence of *n* operations takes *worst-case* time *T*(*n*) in total. In the worst case, the average cost, or *amortized cost*, per operation is therefore *T*(*n*)/*n*.

Aggregate Method



Idea : Show that for all *n*, a sequence of *n* operations takes *worst-case* time *T*(*n*) in total. In the worst case, the average cost, or *amortized cost*, per operation is therefore *T*(*n*)/*n*.

〖Example〗 Stack with MultiPop(int k, Stack S)

Aggregate Method

Idea : Show that for all *n*, a sequence of *n* operations takes *worst-case* time *T*(*n*) in total. In the worst case, the average cost, or *amortized cost*, per operation is therefore *T*(*n*)/*n*.

〚Example〛 Stack with MultiPop(int k, Stack S)

```
Algorithm  {
    while ( !IsEmpty(S) && k>0 ) {
        Pop(S);
        k - -;
    } /* end while-loop */
}
```

Aggregate Method

Idea : Show that for all *n*, a sequence of *n* operations takes *worst-case* time $T(n)$ in total. In the worst case, the average cost, or *amortized cost*, per operation is therefore $T(n)/n$.

〚Example〛 Stack with MultiPop(int k, Stack S)

```
Algorithm  {
    while ( !IsEmpty(S) && k>0 ) {
        Pop(S);
        k - -;
    } /* end while-loop */
}
        T = min ( sizeof(S), k )
```

Aggregate Method

Idea : Show that for all *n*, a sequence of *n* operations takes *worst-case* time $T(n)$ in total. In the worst case, the average cost, or *amortized cost*, per operation is therefore $T(n)/n$.

〚Example〛 Stack with MultiPop(int k, Stack S)

```
Algorithm  {
    while ( !IsEmpty(S) && k>0 ) {
        Pop(S);
        k - -;
    } /* end while-loop */
}
```

$T = \min (\text{sizeof}(S), k)$

Consider a sequence of *n* Push, Pop, and MultiPop operations on an initially empty stack.

Aggregate Method

Idea : Show that for all *n*, a sequence of *n* operations takes *worst-case* time *T*(*n*) in total. In the worst case, the average cost, or *amortized cost*, per operation is therefore *T*(*n*)/*n*.

〚Example〛 Stack with MultiPop(int k, Stack S)

```
Algorithm  {
    while ( !IsEmpty(S) && k>0 ) {
        Pop(S);
        k - -;
    } /* end while-loop */
}
```

$T = \min(\text{sizeof}(S), k)$

Consider a sequence of *n* Push, Pop, and MultiPop operations on an initially empty stack.

$\text{sizeof}(S) \leq n$

Aggregate Method

Idea : Show that for all *n*, a sequence of *n* operations takes *worst-case* time $T(n)$ in total. In the worst case, the average cost, or *amortized cost*, per operation is therefore $T(n)/n$.

Total = $O(n^2)$?

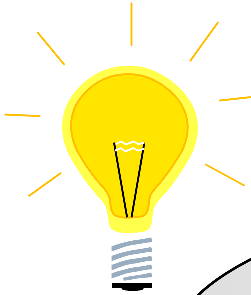
〚Example〛 Stack with MultiPop(int k, Stack S)

```
Algorithm  {
    while ( !IsEmpty(S) && k>0 ) {
        Pop(S);
        k - -;
    } /* end while-loop */
}       $T = \min(\text{sizeof}(S), k)$
```

Consider a sequence of *n* Push, Pop, and MultiPop operations on an initially empty stack.

$\text{sizeof}(S) \le n$

Aggregate Method



Idea : Show that for all n, a sequence of n operations takes *worst-case* time $T(n)$ in total. In the worst case, the average cost, operation is therefore $T(n)/n$.

We can pop each object from the stack *at most once* for each time we have pushed it onto the stack

Total = $O(n^2)$?

〚Example〛 Stack with Multi op(i Stack S)

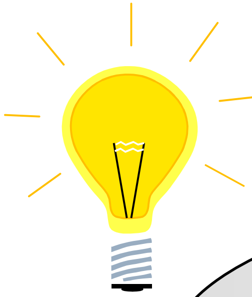
```
Algorithm {
    while ( !IsEmpty(S) && k>0 ) {
        Pop(S);
        k - -;
    } /* end while-loop */
}
```

$T = \min (\text{sizeof}(S), k)$

Consider a sequence of n Push, Pop, and MultiPop operations on an initially empty stack.

$\text{sizeof}(S) \le n$

Aggregate Method



Idea : Show that for all *n*, a sequence of *n* operations takes *worst-case* time $T(n)$ in total. In the worst case, the average cost, or *amortized cost*, per operation is therefore $T(n)/n$.

We can pop each object from the stack *at most once* for each time we have pushed it onto the stack

Total = $O(n^2)$?

〖Example〗 Stack with MultiPop(int k, Stack S)

```
Algorithm  {
    while ( !IsEmpty(S) && k>0 ) {
        Pop(S);
        k - -;
    } /* end while-loop */
}
```

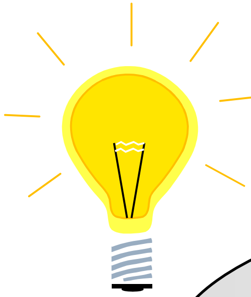
$T = \min(\text{sizeof}(S), k)$

Consider a sequence of *n* Push, Pop, and MultiPop operations on an initially empty stack.

$\text{sizeof}(S) \le n$

$T_{amortized} = O(n)/n = O(1)$

Aggregate Method



Idea : Show that for all *n*, a sequence of *n* operations takes *worst-case* time *T*(*n*) in total. In the worst case, the average cost, per operation is therefore *T*(*n*)/*n*.

We can pop each object from the stack *at most once* for each time we have pushed it onto the stack

Total = *O*(n^2) ?

〚Example〛 Stack with Multi op(Stack S)

```
Algorithm {
    while ( !IsEmpty(S) && k>0 ) {
        Pop(S);
        k - -;
    } /* end while-loop */
}
```

$T = \min(\text{sizeof}(S), k)$

Consider a sequence of *n* Push, Pop, and MultiPop operations on an initially empty stack.

$\text{sizeof}(S) \le n$

$T_{amortized} = O(n)/n = O(1)$

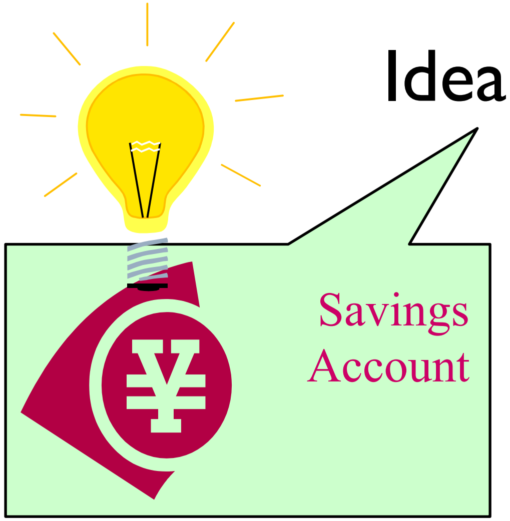
The total time of pop should be less than the total time of push.
The total time of push takes at most O(n).

Accounting Method

Accounting Method

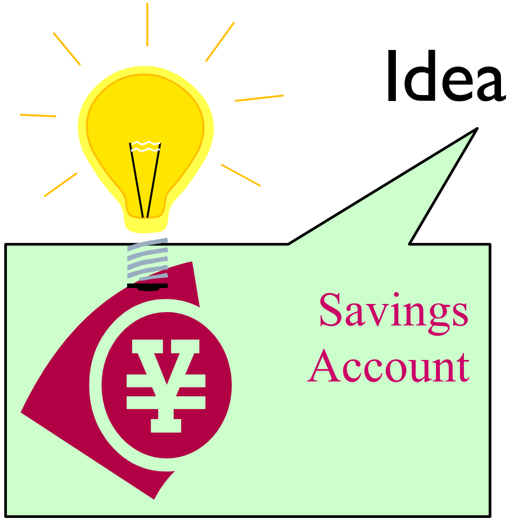
Idea : When an operation's *amortized cost* $\hat{c}_i$ exceeds its *actual cost* c_i , we assign the difference to specific objects in the data structure as *credit*. Credit can help *pay* for later operations whose amortized cost is less than their actual cost.

Accounting Method



Idea : When an operation's *amortized cost* $\hat{c}_i$ exceeds its *actual cost* c_i, we assign the difference to specific objects in the data structure as *credit*. Credit can help *pay* for later operations whose amortized cost is less than their actual cost.

Accounting Method

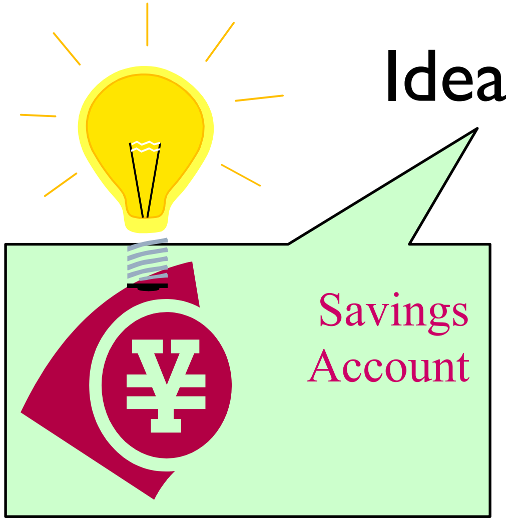


Idea : When an operation's *amortized cost* $\hat{c}_i$ exceeds its *actual cost* c_i, we assign the difference to specific objects in the data structure as *credit*. Credit can help *pay* for later operations whose amortized cost is less than their actual cost.

Note: For all sequences of *n* operations, we must have

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

Accounting Method



Idea : When an operation's *amortized cost* $\hat{c}_i$ exceeds its *actual cost* c_i, we assign the difference to specific objects in the data structure as *credit*. Credit can help *pay* for later operations whose amortized cost is less than their actual cost.

Note: For all sequences of *n* operations, we must have

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

$$T_{amortized} = \frac{1}{n} \sum_{i}^{n} \hat{c}_i$$

〖Example〗 Stack with MultiPop(int k, Stack S)

〖Example〗 Stack with MultiPop(int k, Stack S)

c_i for Push: ; Pop: ; and MultiPop:

〖Example〗 Stack with MultiPop(int k, Stack S)

c_i for Push: 1 ; Pop: ; and MultiPop:

〖Example〗 Stack with MultiPop(int k, Stack S)

c_i for Push: 1 ; Pop: 1 ; and MultiPop:

〖Example〗 Stack with MultiPop(int k, Stack S)

c_i for Push: 1 ; Pop: 1 ; and MultiPop: $\min(\text{sizeof}(S), k)$

〖Example〗 Stack with MultiPop(int k, Stack S)

c_i for Push: 1 ; Pop: 1 ; and MultiPop: $\min(\text{sizeof}(S), k)$

$\hat{c}_i$ for Push: ; Pop: ; and MultiPop:

〖Example〗 Stack with MultiPop(int k, Stack S)

c_i for Push: 1 ; Pop: 1 ; and MultiPop: $\min(\text{sizeof}(S), k)$

$\hat{c}_i$ for Push: 2 ; Pop: ; and MultiPop:

〖Example〗 Stack with MultiPop(int k, Stack S)

c_i for Push: 1 ; Pop: 1 ; and MultiPop: $\min(\text{sizeof}(S), k)$

$\hat{c}_i$ for Push: 2 ; Pop: 0 ; and MultiPop:

〖Example〗 Stack with MultiPop(int k, Stack S)

c_i for Push: 1 ; Pop: 1 ; and MultiPop: $\min(\text{sizeof}(S), k)$

$\hat{c}_i$ for Push: 2 ; Pop: 0 ; and MultiPop: 0

【Example】 Stack with MultiPop(int k, Stack S)

c_i for Push: 1 ; Pop: 1 ; and MultiPop: $\min(\text{sizeof}(S), k)$

$\hat{c}_i$ for Push: 2 ; Pop: 0 ; and MultiPop: 0

Starting from an empty stack —— *Credits* for

〖Example〗 Stack with MultiPop(int k, Stack S)

c_i for Push: 1 ; Pop: 1 ; and MultiPop: $\min(\,\text{sizeof}(S),\, k\,)$

$\hat{c}_i$ for Push: 2 ; Pop: 0 ; and MultiPop: 0

Starting from an empty stack —— *Credits* for

Push: ; Pop: ; and MultiPop:

〖Example〗 Stack with MultiPop(int k, Stack S)

c_i for Push: 1 ; Pop: 1 ; and MultiPop: $\min(\text{sizeof}(S), k)$

$\hat{c}_i$ for Push: 2 ; Pop: 0 ; and MultiPop: 0

Starting from an empty stack —— *Credits* for

Push: +1 ; Pop: ; and MultiPop:

〖Example〗 Stack with MultiPop(int k, Stack S)

c_i for Push: 1 ; Pop: 1 ; and MultiPop: $\min(\text{sizeof}(S), k)$

$\hat{c}_i$ for Push: 2 ; Pop: 0 ; and MultiPop: 0

Starting from an empty stack —— *Credits* for

Push: +1 ; Pop: −1 ; and MultiPop:

〖Example〗 Stack with MultiPop(int k, Stack S)

c_i for Push: 1 ; Pop: 1 ; and MultiPop: $\min(\text{sizeof}(S), k)$

$\hat{c}_i$ for Push: 2 ; Pop: 0 ; and MultiPop: 0

Starting from an empty stack —— *Credits* for

Push: +1 ; Pop: –1 ; and MultiPop: –1 for each +1

〖Example〗 Stack with MultiPop(int k, Stack S)

c_i for Push: 1 ; Pop: 1 ; and MultiPop: $\min(\text{sizeof}(S), k)$

$\hat{c}_i$ for Push: 2 ; Pop: 0 ; and MultiPop: 0

Starting from an empty stack —— *Credits* for

Push: +1 ; Pop: −1 ; and MultiPop: −1 for each +1

sizeof(*S*) ≥ 0 ➡ *Credits* ≥ 0

〚Example〛 Stack with MultiPop(int k, Stack S)

c_i for Push: 1 ; Pop: 1 ; and MultiPop: $\min(\text{sizeof}(S), k)$

$\hat{c}_i$ for Push: 2 ; Pop: 0 ; and MultiPop: 0

Starting from an empty stack —— *Credits* for

Push: +1 ; Pop: −1 ; and MultiPop: −1 for each +1

sizeof(*S*) ≥ 0 ➡ *Credits* ≥ 0

$$\Rightarrow \ O(n) = \sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

〖Example〗 Stack with MultiPop(int k, Stack S)

c_i for Push: 1 ; Pop: 1 ; and MultiPop: $\min(\text{sizeof}(S), k)$

$\hat{c}_i$ for Push: 2 ; Pop: 0 ; and MultiPop: 0

Starting from an empty stack —— *Credits* for

Push: +1 ; Pop: −1 ; and MultiPop: −1 for each +1

sizeof(*S*) ≥ 0 ➡ *Credits* ≥ 0

➡ $$O(n) = \sum_{i=1}^{n} \hat{c}_i \ge \sum_{i=1}^{n} c_i$$

➡ $T_{amortized} = O(\,n\,)/n = O(1)$

〖Example〗 Stack with MultiPop(int k, Stack S)

c_i for Push: 1 ; Pop: 1 ; and MultiPop: $\min(\text{sizeof}(S), k)$

$\hat{c}_i$ for Push: 2 ; Pop: 0 ; and MultiPop: 0

Starting from an stack —— Cre

Push: +1 ; Pop: –1

sizeof(S) ≥ 0 ➡ Cre

The amortized costs of the operations may *differ* from each other

➡ $O(n) = \sum_{i=1} c_i \quad \sum_{i=1} c_i$

➡ $T_{amortized} = O(n)/n = O(1)$

Potential Method

- Why some problems have smaller amortized time cost?
  - The structure of the problem provides the constraints:

- Represent the states of the structure as potential functions.
  - The potential function is bounded by the structural constraints.
  - Bound the total cost by the increase of potential.

Potential Method

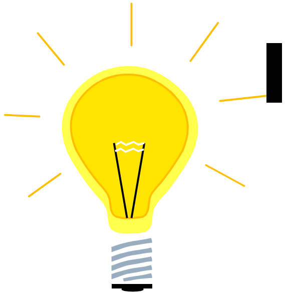
- Why some problems have smaller amortized time cost?
  - The structure of the problem provides the constraints:

All operations can not exceed the structural constraints.

- Represent the states of the structure as potential functions.
  - The potential function is bounded by the structural constraints.
  - Bound the total cost by the increase of potential.

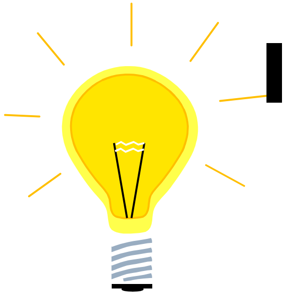
Potential Method

Potential Method



Idea : Take a closer look at the *credit* --

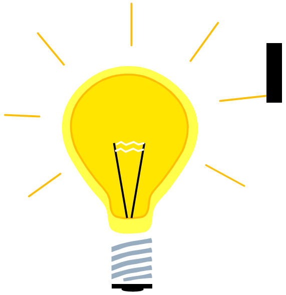
Potential Method



Idea : Take a closer look at the *credit* --

$$\hat{c}_i - c_i = Credit_i = \Phi(D_i) - \Phi(D_{i-1})$$

Potential Method



Idea : Take a closer look at the *credit* --

$$\hat{\boldsymbol{c}}_i - \boldsymbol{c}_i = \boldsymbol{Credit}_i = \Phi(\boldsymbol{D}_i) - \Phi(\boldsymbol{D}_{i-1})$$

Potential function

Potential Method

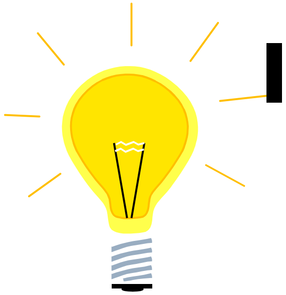
Idea : Take a closer look at the *credit* --

$$\hat{c}_i - c_i = Credit_i = \Phi(D_i) - \Phi(D_{i-1})$$

Potential function

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1}) \right)$$

Potential Method



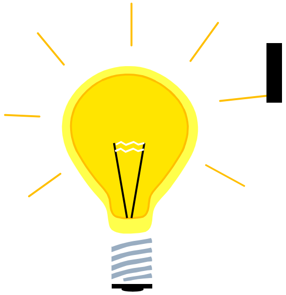
Idea : Take a closer look at the *credit* --

$$\hat{c}_i - c_i = \mathit{Credit}_i = \Phi(D_i) - \Phi(D_{i-1})$$

Potential function

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right) \\ &= \left(\sum_{i=1}^{n} c_i\right) + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

Potential Method



Idea : Take a closer look at the *credit* --

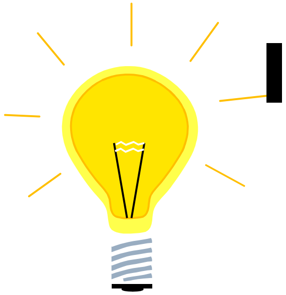
$$\hat{c}_i - c_i = Credit_i = \Phi(D_i) - \Phi(D_{i-1})$$

Potential function

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1}) \right)$$

$$= \left(\sum_{i=1}^{n} c_i \right) + \underbrace{\Phi(D_n) - \Phi(D_0)}_{\geq 0}$$

Potential Method



Idea : Take a closer look at the *credit* --

$$\hat{c}_i - c_i = \boldsymbol{Credit}_i = \Phi(D_i) - \Phi(D_{i-1})$$

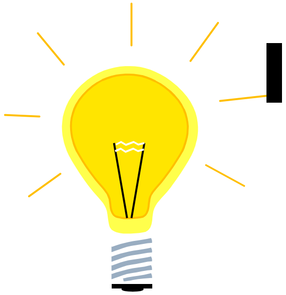
Potential function

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1}) \right)$$

$$= \left(\sum_{i=1}^{n} c_i \right) + \underbrace{\Phi(D_n) - \Phi(D_0)}_{\geq 0}$$

In general, a good potential function should always assume its minimum at the start of the sequence.

Potential Method



Idea : Take a closer look at the *credit* --

$$\hat{c}_i - c_i = \boldsymbol{Credit}_i = \Phi(D_i) - \Phi(D_{i-1})$$

Potential function

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1}) \right)$$

$$= \left(\sum_{i=1}^{n} c_i \right) + \underbrace{\Phi(D_n) - \Phi(D_0)}_{\geq 0}$$

Should be bounded.

In general, a good potential function should always assume its minimum at the start of the sequence.

〖Example〗 Stack with MultiPop(int k, Stack S)

〖Example〗 Stack with MultiPop(int k, Stack S)

D_i =

$\Phi(D_i)$ =

〖Example〗 Stack with MultiPop(int k, Stack S)

D_i = the stack that results after the *i*-th operation

$\Phi(D_i)$ =

〖Example〗 Stack with MultiPop(int k, Stack S)

D_i = the stack that results after the *i*-th operation

$\Phi(D_i)$ = the number of objects in the stack D_i

〖Example〗 Stack with MultiPop(int k, Stack S)

D_i = the stack that results after the *i*-th operation

$\Phi(D_i)$ = the number of objects in the stack D_i

$$\Phi(D_i) \geq 0 = \Phi(D_0)$$

〖Example〗 Stack with MultiPop(int k, Stack S)

D_i = the stack that results after the *i*-th operation

$\Phi(D_i)$ = the number of objects in the stack D_i

$$\Phi(D_i) \geq 0 = \Phi(D_0)$$

Push:

〖Example〗 Stack with MultiPop(int k, Stack S)

D_i = the stack that results after the *i*-th operation

Φ(D_i) = the number of objects in the stack D_i

$$\Phi(D_i) \geq 0 = \Phi(D_0)$$

Push: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S) + 1) - sizeof(S) = 1$

〚Example〛 Stack with MultiPop(int k, Stack S)

D_i = the stack that results after the *i*-th operation

$\Phi(D_i)$ = the number of objects in the stack D_i

$$\Phi(D_i) \geq 0 = \Phi(D_0)$$

Push: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S) + 1) - sizeof(S) = 1$

➡ $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$

〖Example〗 Stack with MultiPop(int k, Stack S)

D_i = the stack that results after the *i*-th operation

$\Phi(D_i)$ = the number of objects in the stack D_i

$$\Phi(D_i) \geq 0 = \Phi(D_0)$$

Push: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S) + 1) - sizeof(S) = 1$

➡ $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$

Pop:

〖Example〗 Stack with MultiPop(int k, Stack S)

D_i = the stack that results after the *i*-th operation

$\Phi(D_i)$ = the number of objects in the stack D_i

$$\Phi(D_i) \geq 0 = \Phi(D_0)$$

Push: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S) + 1) - sizeof(S) = 1$

$\Rightarrow \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$

Pop: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S) - 1) - sizeof(S) = -1$

〖Example〗 Stack with MultiPop(int k, Stack S)

D_i = the stack that results after the *i*-th operation

$\Phi(D_i)$ = the number of objects in the stack D_i

$$\Phi(D_i) \geq 0 = \Phi(D_0)$$

Push: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S) + 1) - sizeof(S) = 1$

➡ $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$

Pop: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S) - 1) - sizeof(S) = -1$

➡ $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0$

〖Example〗 Stack with MultiPop(int k, Stack S)

D_i = the stack that results after the *i*-th operation

$\Phi(D_i)$ = the number of objects in the stack D_i

$$\Phi(D_i) \geq 0 = \Phi(D_0)$$

Push: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S)+1) - sizeof(S) = 1$

➡ $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$

Pop: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S)-1) - sizeof(S) = -1$

➡ $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0$

MultiPop:

〚Example〛 Stack with MultiPop(int k, Stack S)

D_i = the stack that results after the *i*-th operation

$\Phi(D_i)$ = the number of objects in the stack D_i

$$\Phi(D_i) \geq 0 = \Phi(D_0)$$

Push: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S) + 1) - sizeof(S) = 1$

➡ $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$

Pop: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S) - 1) - sizeof(S) = -1$

➡ $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0$

MultiPop: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S) - k') - sizeof(S) = -k'$

〖Example〗 Stack with MultiPop(int k, Stack S)

D_i = the stack that results after the *i*-th operation

$\Phi(D_i)$ = the number of objects in the stack D_i

$$\Phi(D_i) \geq 0 = \Phi(D_0)$$

Push: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S) + 1) - sizeof(S) = 1$

➡ $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$

Pop: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S) - 1) - sizeof(S) = -1$

➡ $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0$

MultiPop: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S) - k') - sizeof(S) = -k'$

➡ $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0$

〖Example〗 Stack with MultiPop(int k, Stack S)

D_i = the stack that results after the *i*-th operation

$\Phi(D_i)$ = the number of objects in the stack D_i

$$\Phi(D_i) \geq 0 = \Phi(D_0)$$

Push: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S) + 1) - sizeof(S) = 1$

➡ $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$

Pop: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S) - 1) - sizeof(S) = -1$

➡ $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0$

MultiPop: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S) - k') - sizeof(S) = -k'$

➡ $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0$

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} O(1) = O(n)$$

〚Example〛 Stack with MultiPop(int k, Stack S)

D_i = the stack that results after the *i*-th operation

$\Phi(D_i)$ = the number of objects in the stack D_i

$$\Phi(D_i) \geq 0 = \Phi(D_0)$$

Push: $\Phi(D_i) - \Phi(D_{i-1}) = (\mathit{sizeof}(S) + 1) - \mathit{sizeof}(S) = 1$

➡ $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$

Pop: $\Phi(D_i) - \Phi(D_{i-1}) = (\mathit{sizeof}(S) - 1) - \mathit{sizeof}(S) = -1$

➡ $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0$

MultiPop: $\Phi(D_i) - \Phi(D_{i-1}) = (\mathit{sizeof}(S) - k') - \mathit{sizeof}(S) = -k'$

➡ $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0$

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} O(1) = O(n) \geq \sum_{i=1}^{n} c_i$$

〖Example〗 Stack with MultiPop(int k, Stack S)

D_i = the stack that results after the i-th operation

$\Phi(D_i)$ = the number of objects in the stack D_i

$$\Phi(D_i) \geq 0 = \Phi(D_0)$$

Push: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S)+1) - sizeof(S) = 1$

$$\Rightarrow \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$$

Pop: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S)-1) - sizeof(S) = -1$

$$\Rightarrow \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0$$

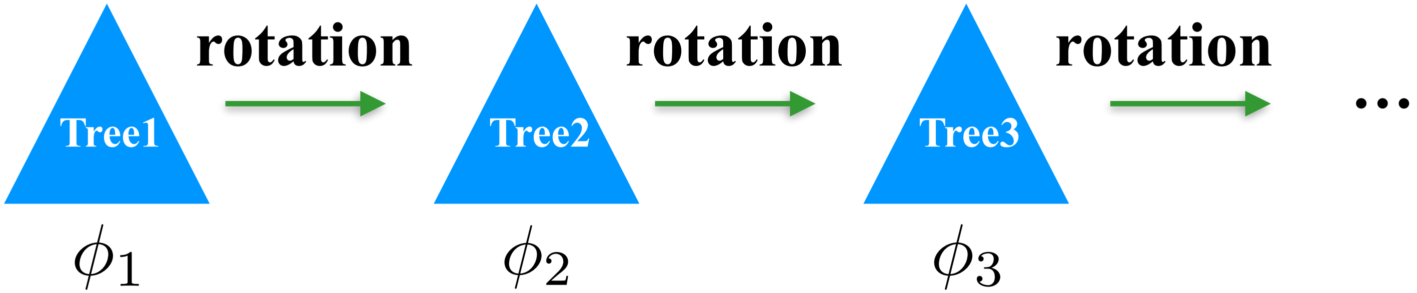
MultiPop: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S) - k') - sizeof(S) = -k'$

$$\Rightarrow \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0$$

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} O(1) = O(n) \geq \sum_{i=1}^{n} c_i \Rightarrow$$ $T_{amortized} = O(n)/n = O(1)$

Analysis of Splay Trees

- What we want to bound?
  - The amortized cost of a sequence of operations, e.g. search, delete, insert, split…
  - Each operation involves slaying: a subsequence of rotations.
- The potential function is built on a state of tree. Let's consider the amortized cost of sequence of rotations first.



〖Example〗 Splay Trees: $T_{amortized} = O(\log N)$

〖Example〗 Splay Trees: $T_{amortized} = O(\log N)$

$D_i =$

$\Phi(D_i) =$

〖Example〗 Splay Trees: $T_{amortized} = O(\log N)$

D_i = the root of the resulting tree

$\Phi(D_i)$ =

〖Example〗 Splay Trees: $T_{amortized} = O(\log N)$

D_i = the root of the resulting tree

$\Phi(D_i)$ = must increase by at most $O(\log N)$ over n steps, AND will also cancel out the number of rotations (*zig*:1; *zig-zag*:2; *zig-zig*:2).

〖Example〗 Splay Trees: $T_{amortized} = O(\log N)$

D_i = the root of the resulting tree

$\Phi(D_i)$ = must increase by at most $O(\log N)$ over n steps, AND will also cancel out the number of rotations (*zig*:1; *zig-zag*:2; *zig-zig*:2).

$$\Phi(T) = \sum_{i \in T} \log S(i)$$

where $S(i)$ is the number of descendants of i (i included).

〖Example〗 Splay Trees: $T_{amortized} = O(\log N)$

D_i = the root of the resulting tree

$\Phi(D_i)$ = must increase by at most $O(\log N)$ over n steps, AND will also cancel out the number of rotations (*zig*:1; *zig-zag*:2; *zig-zig*:2).

$$\Phi(T) = \sum_{i \in T} \log S(i)$$

where $S(i)$ is the number of descendants of i (i included).

Rank of the subtree
≈ Height of the tree

〖Example〗 Splay Trees: $T_{amortized} = O(\log N)$

D_i = the root of the resulting tree

$\Phi(D_i)$ = must increase by at most $O(\log N)$ over n steps, AND will also cancel out the number of rotations (*zig*:1; *zig-zag*:2; *zig-zig*:2).

$$\Phi(T) = \sum_{i \in T} \log S(i)$$

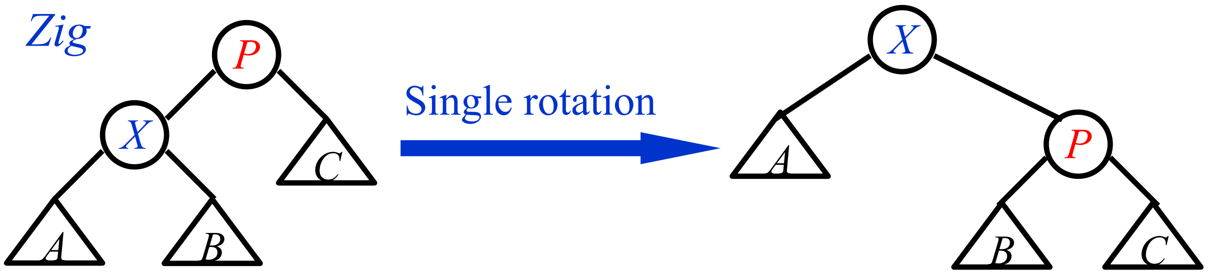
where $S(i)$ is the number of descendants of i (i included).

Rank of the subtree
≈ Height of the tree

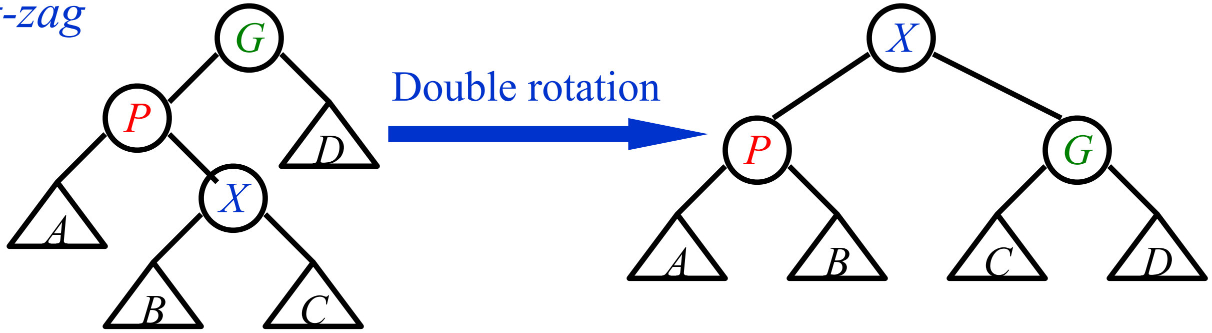
Why not simply use the heights of the trees?

$$\Phi(T) = \sum_{i \in T} Rank(i)$$

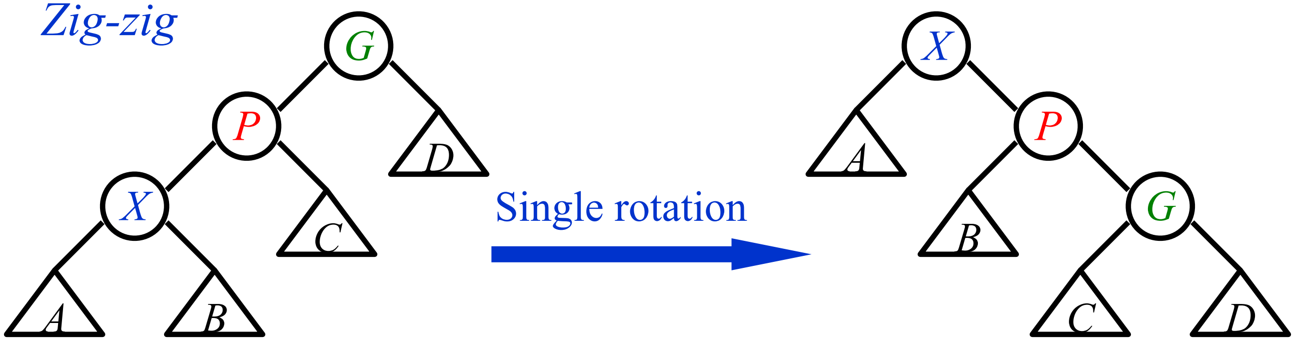
Zig


Single rotation

Zig-zag

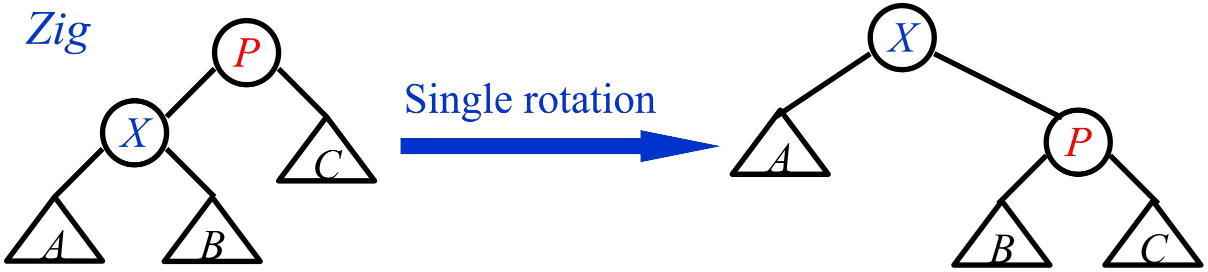

Double rotation

Zig-zig


Single rotation

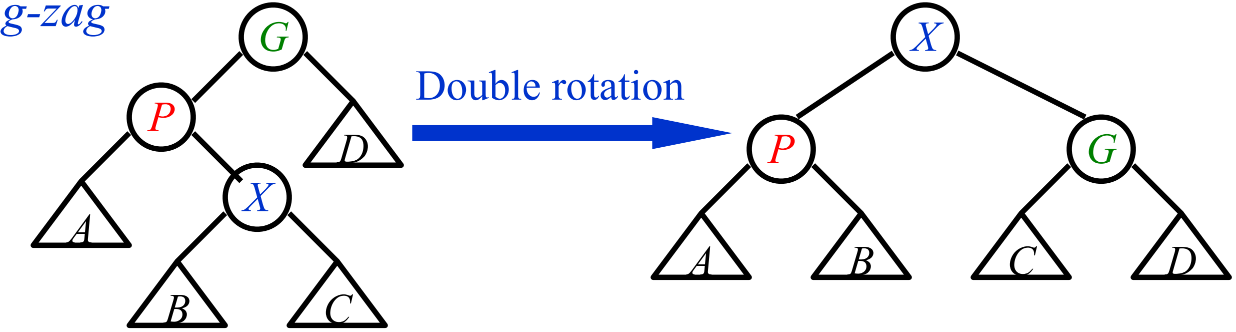
$$\Phi(T) = \sum_{i \in T} Rank(i)$$

Zig

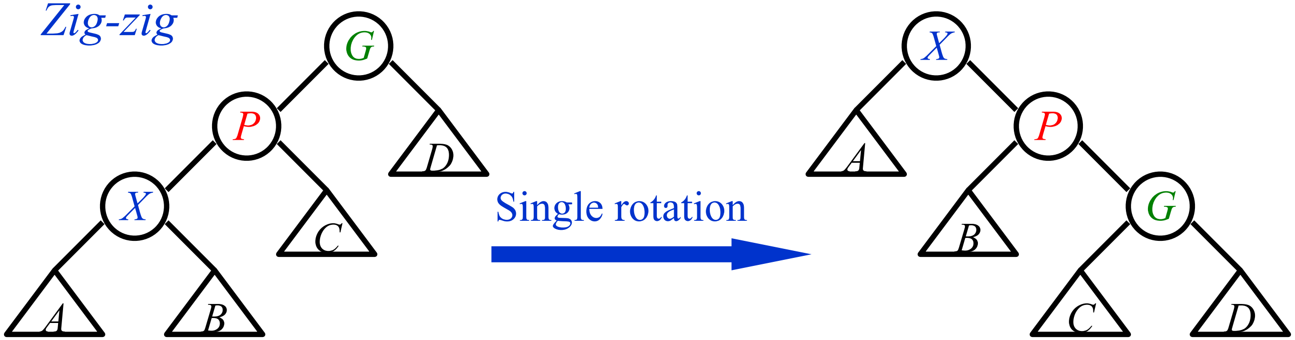


$$\begin{aligned} \hat{c}_i &= 1 + R_2(X) - R_1(X) \\ &\quad + R_2(P) - R_1(P) \\ &\le 1 + R_2(X) - R_1(X) \end{aligned}$$

Zig-zag

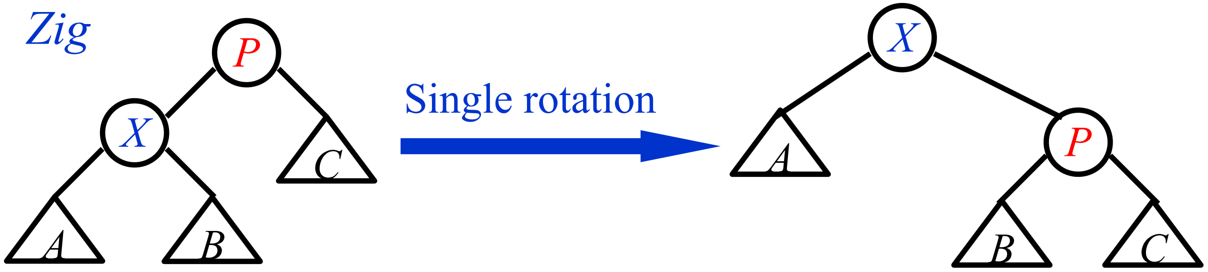


Zig-zig



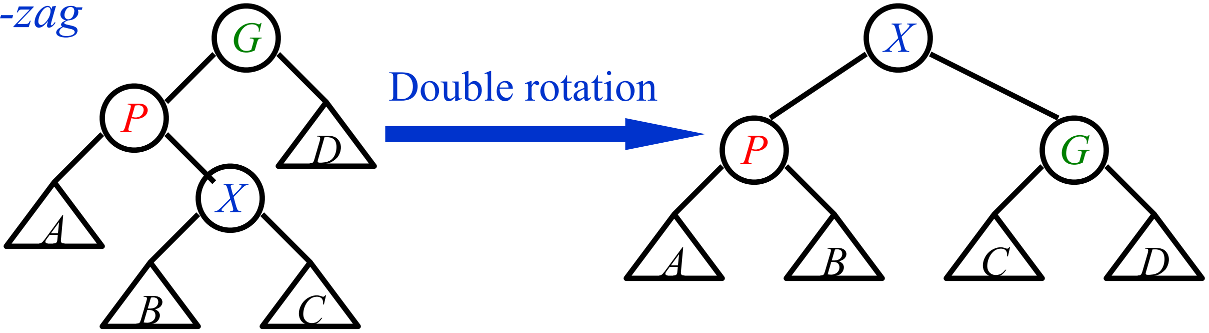
$$\Phi(T) = \sum_{i \in T} Rank(i)$$

Zig

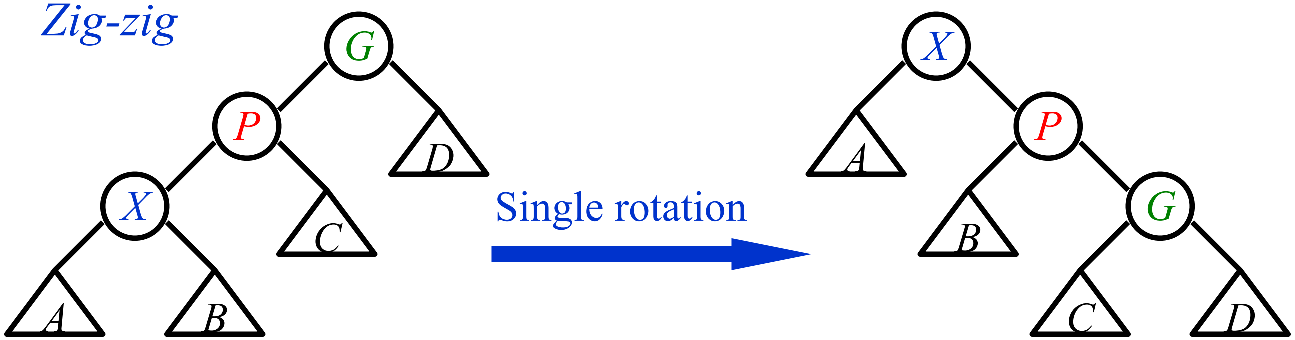


$$\begin{aligned} \hat{c}_i &= 1 + R_2(X) - R_1(X) \\ &\quad + \underline{R_2(P) - R_1(P)} \\ &\le 1 + R_2(X) - R_1(X) \end{aligned}$$

Zig-zag

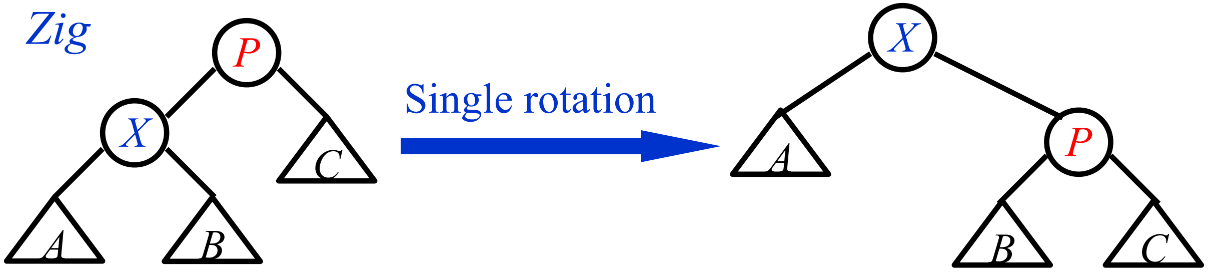


Zig-zig



$$\Phi(T) = \sum_{i \in T} Rank(i)$$

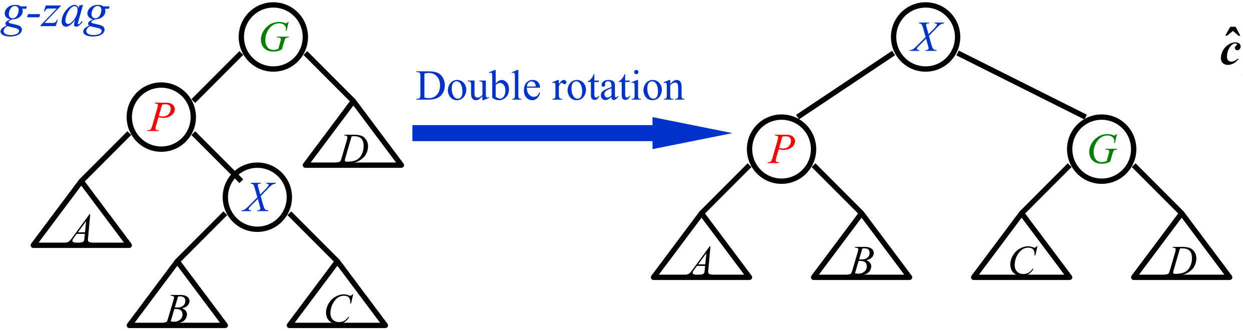
Zig



Single rotation

$$\begin{aligned}\hat{c}_i &= 1 + R_2(X) - R_1(X) \\ &\quad + \underline{R_2(P) - R_1(P)} \\ &\le 1 + R_2(X) - R_1(X)\end{aligned}$$

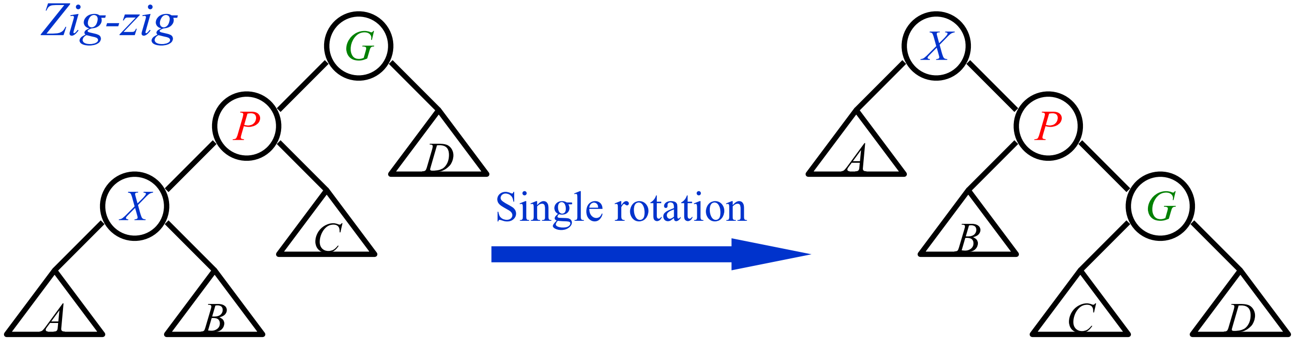
Zig-zag



Double rotation

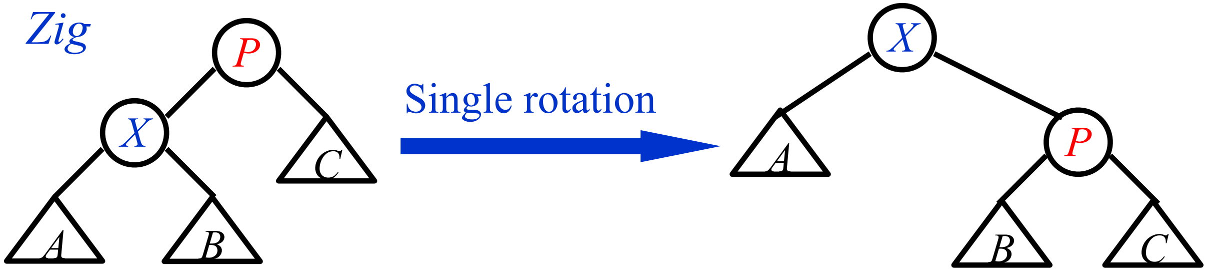
$$\begin{aligned}\hat{c}_i &= 2 + R_2(X) - R_1(X) \\ &\quad + R_2(P) - R_1(P) \\ &\quad + R_2(G) - R_1(G) \\ &\le 2\big(R_2(X) - R_1(X)\big)\end{aligned}$$

Zig-zig

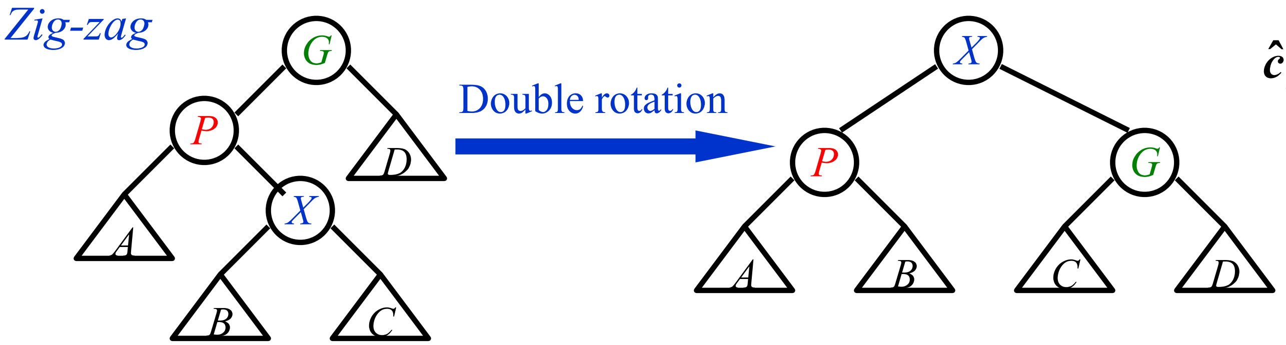


Single rotation

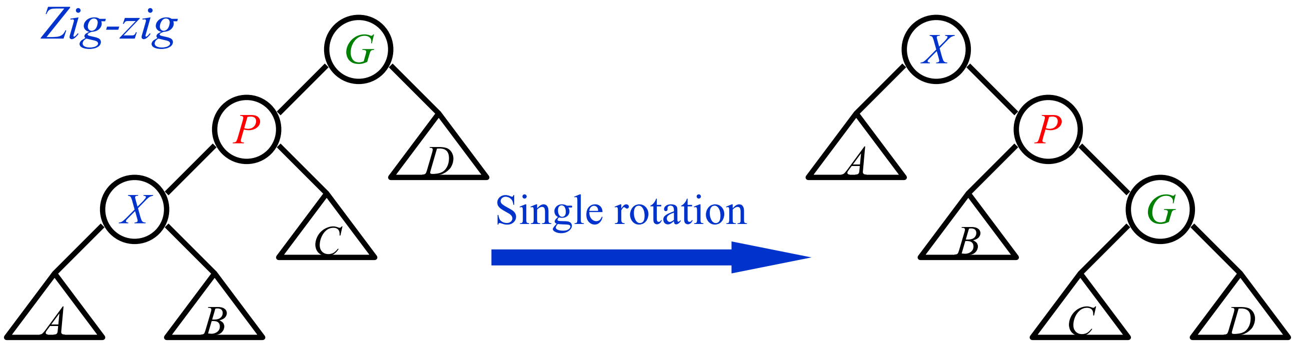
$$\Phi(T) = \sum_{i\in T} Rank(i)$$



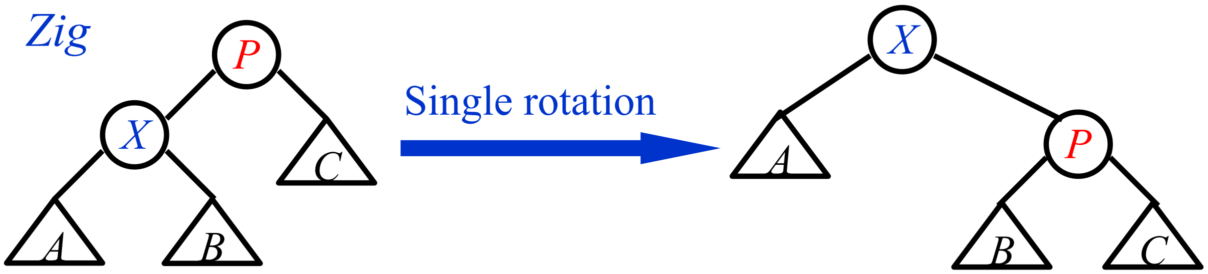
$$\hat{c}_i = 1 + R_2(X) - R_1(X)$$
$$+ \underline{R_2(P) - R_1(P)}$$
$$\leq 1 + R_2(X) - R_1(X)$$



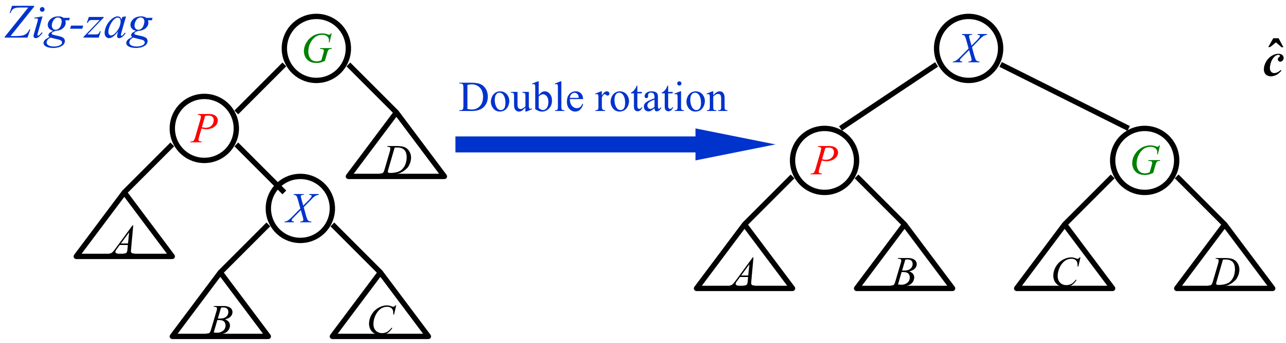
$$\hat{c}_i = 2 + \cancel{R_2(X)} - R_1(X)$$
$$+ R_2(P) - R_1(P)$$
$$+ R_2(G) - \cancel{R_1(G)}$$
$$\leq 2\big(R_2(X) - R_1(X)\big)$$



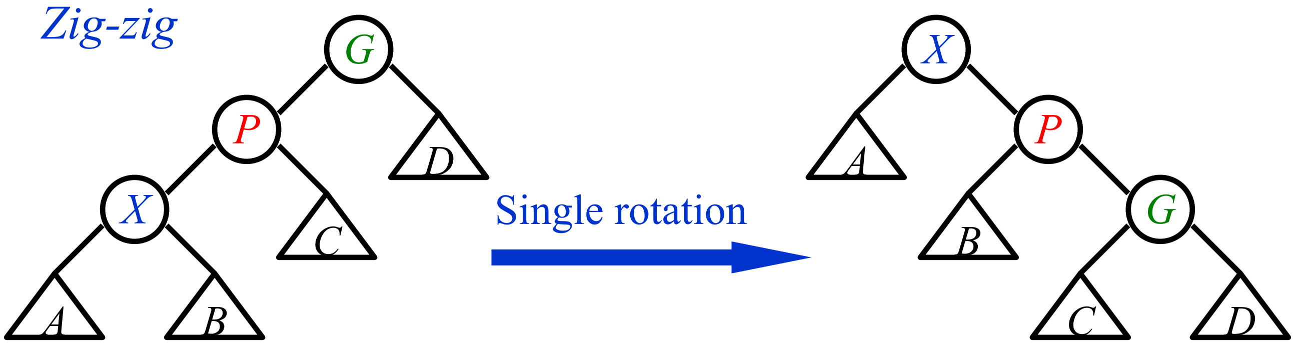
$$\Phi(T) = \sum_{i \in T} Rank(i)$$



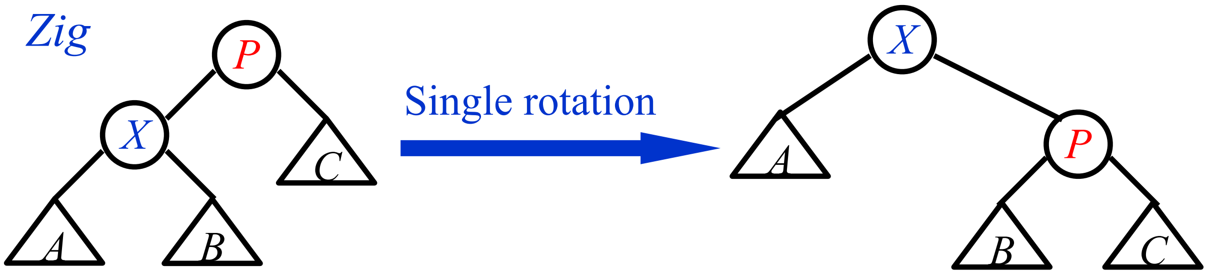
$$\begin{aligned} \hat{c}_i &= 1 + R_2(X) - R_1(X) \\ &\quad + \underline{R_2(P) - R_1(P)} \\ &\le 1 + R_2(X) - R_1(X) \end{aligned}$$



$$\begin{aligned} \hat{c}_i &= 2 + \cancel{R_2(X)} - \underline{R_1(X)} \\ &\quad + R_2(P) - \underline{R_1(P)} \\ &\quad + R_2(G) - \cancel{R_1(G)} \\ &\le 2\left(R_2(X) - \underline{R_1(X)}\right) \end{aligned}$$

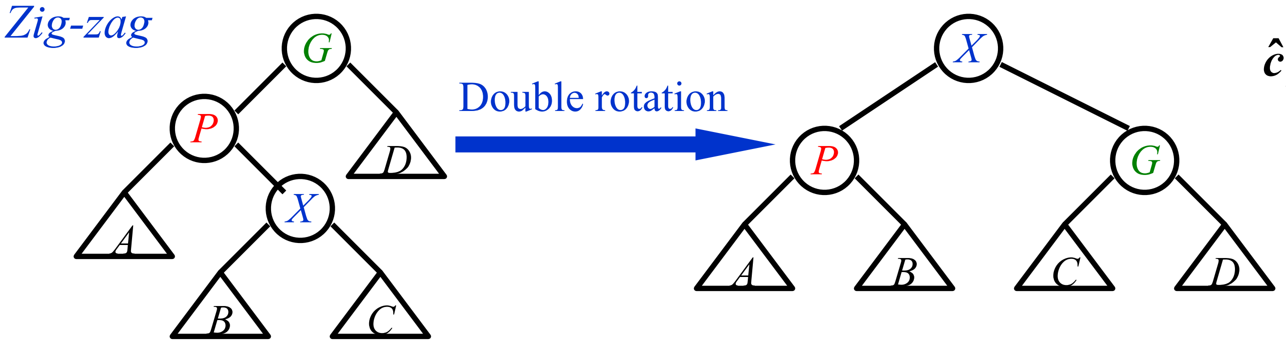


$$\Phi(T) = \sum_{i \in T} Rank(i)$$


Zig

Single rotation

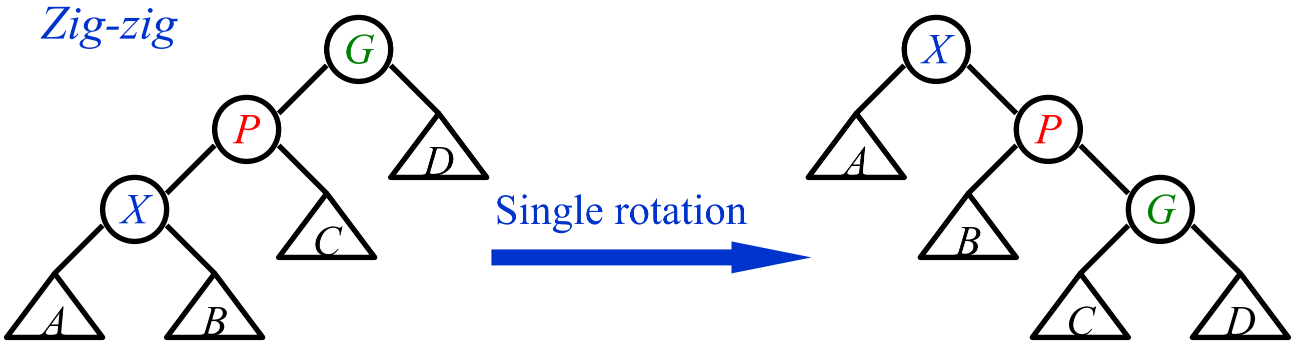
$$\begin{aligned}\hat{c}_i &= 1 + R_2(X) - R_1(X)\\ &\quad + \underline{R_2(P) - R_1(P)}\\ &\le 1 + R_2(X) - R_1(X)\end{aligned}$$


Zig-zag

Double rotation

$$\begin{aligned}\hat{c}_i &= 2 + \cancel{R_2(X)} - \underline{R_1(X)}\\ &\quad + R_2(P) - \underline{R_1(P)}\\ &\quad + R_2(G) - \cancel{R_1(G)}\\ &\le 2\left(R_2(X) - \underline{R_1(X)}\right)\end{aligned}$$

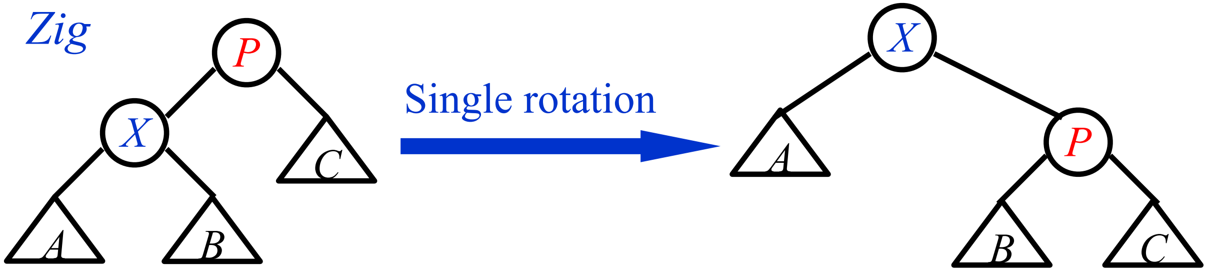
Lemma 11.4 on [Weiss] p.448


Zig-zig

Single rotation

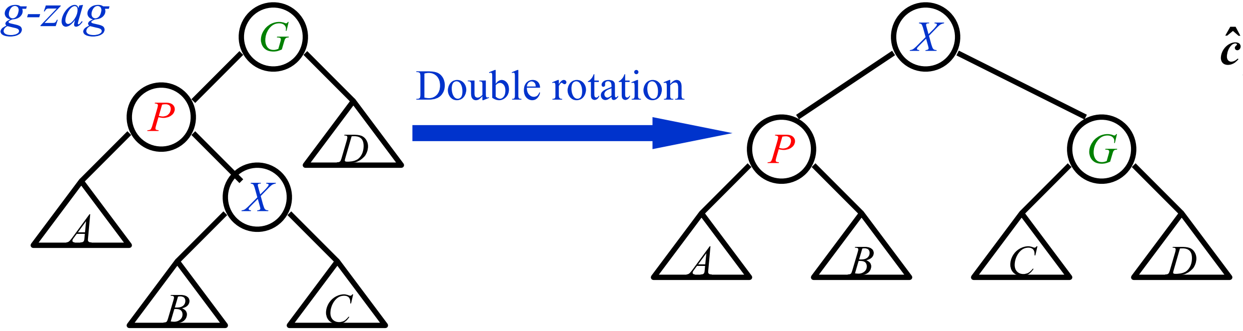
$$\Phi(T) = \sum_{i \in T} Rank(i)$$

Zig


Single rotation

$$\begin{aligned}\hat{c}_i &= 1 + R_2(X) - R_1(X) \\ &\quad + \underline{R_2(P) - R_1(P)} \\ &\le 1 + R_2(X) - R_1(X)\end{aligned}$$

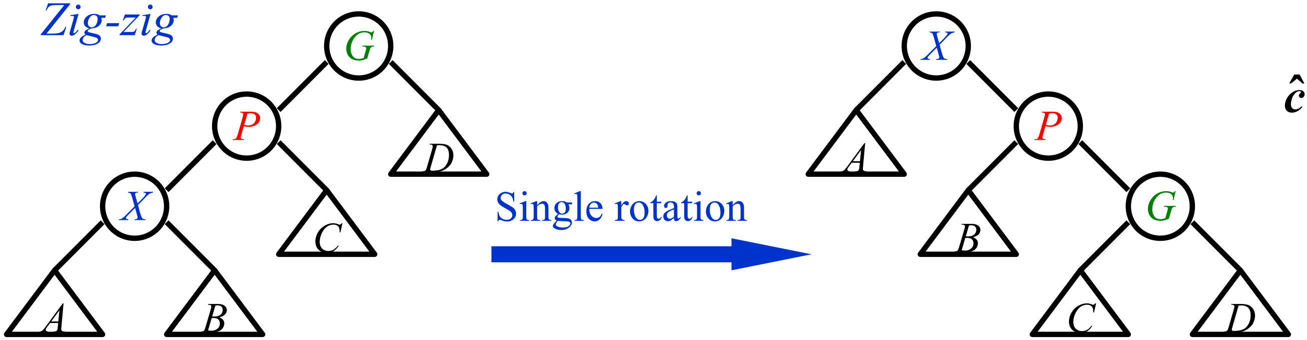
Zig-zag


Double rotation

$$\begin{aligned}\hat{c}_i &= 2 + \cancel{R_2(X)} - \underline{R_1(X)} \\ &\quad + R_2(P) - \underline{R_1(P)} \\ &\quad + R_2(G) - \cancel{R_1(G)} \\ &\le 2\left(R_2(X) - \underline{R_1(X)}\right)\end{aligned}$$

Lemma 11.4 on [Weiss] p.448

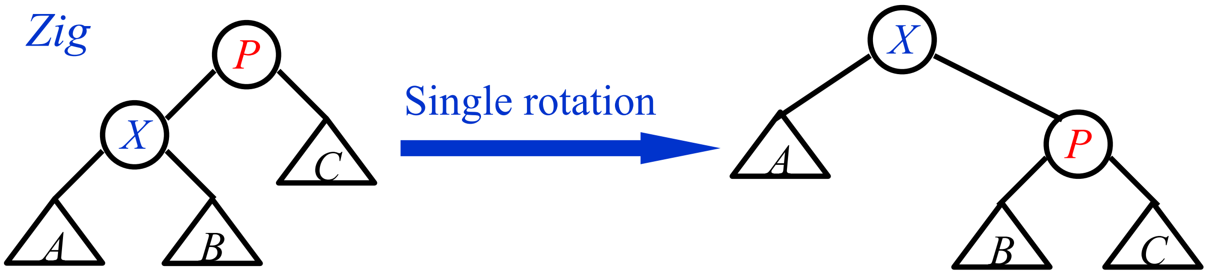
Zig-zig


Single rotation

$$\begin{aligned}\hat{c}_i &= 2 + R_2(X) - R_1(X) \\ &\quad + R_2(P) - R_1(P) \\ &\quad + R_2(G) - R_1(G) \\ &\le 3\left(R_2(X) - R_1(X)\right)\end{aligned}$$

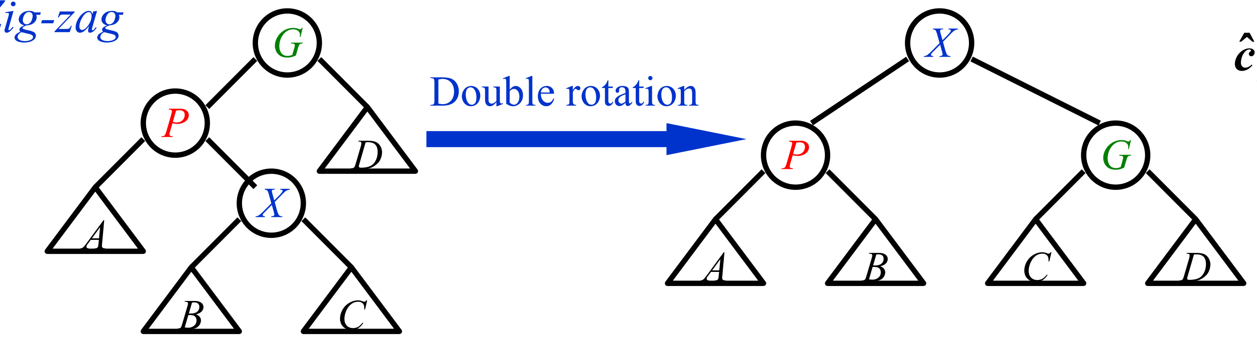
$$\Phi(T) = \sum_{i\in T} Rank(i)$$

Zig



$$\hat{c}_i = 1 + R_2(X) - R_1(X) + \underline{R_2(P) - R_1(P)} \le 1 + R_2(X) - R_1(X)$$

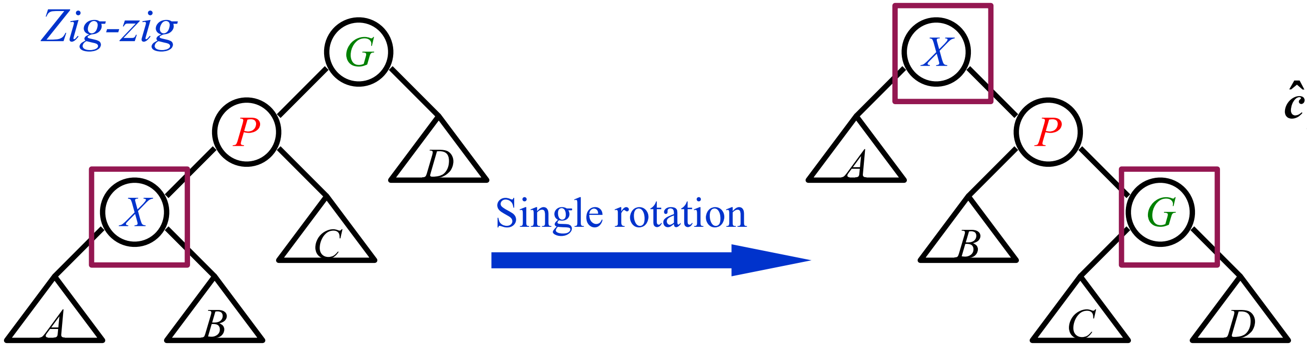
Zig-zag



$$\hat{c}_i = 2 + \cancel{R_2(X)} - \underline{R_1(X)} + R_2(P) - \underline{R_1(P)} + R_2(G) - \cancel{R_1(G)} \le 2\left(R_2(X) - \underline{R_1(X)}\right)$$

Lemma 11.4 on [Weiss] p.448

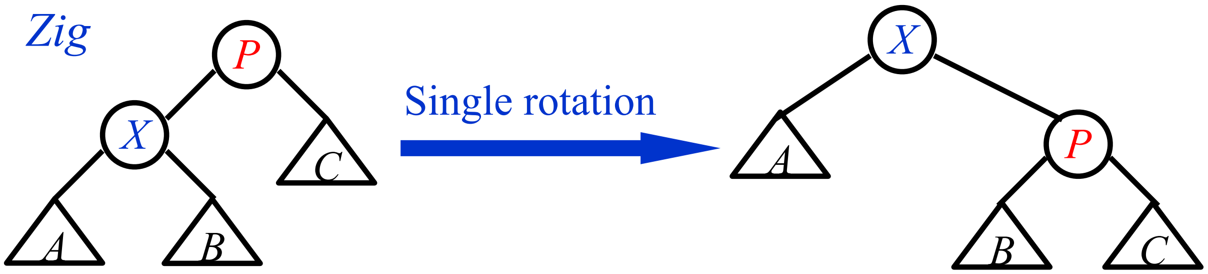
Zig-zig



$$\hat{c}_i = 2 + R_2(X) - R_1(X) + R_2(P) - R_1(P) + R_2(G) - R_1(G) \le 3\left(R_2(X) - R_1(X)\right)$$

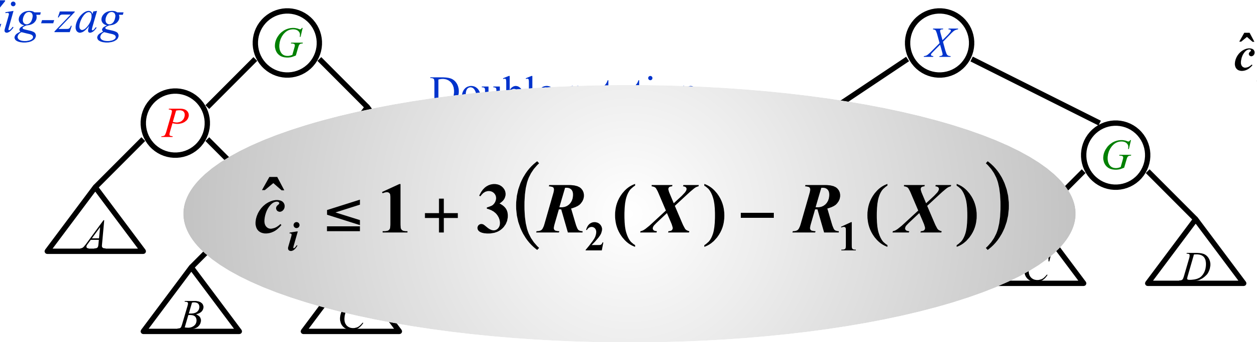
$$\Phi(T) = \sum_{i \in T} Rank(i)$$

Zig



$$\begin{aligned}\hat{c}_i &= 1 + R_2(X) - R_1(X) \\ &\quad + \underline{R_2(P) - R_1(P)} \\ &\le 1 + R_2(X) - R_1(X)\end{aligned}$$

Zig-zag

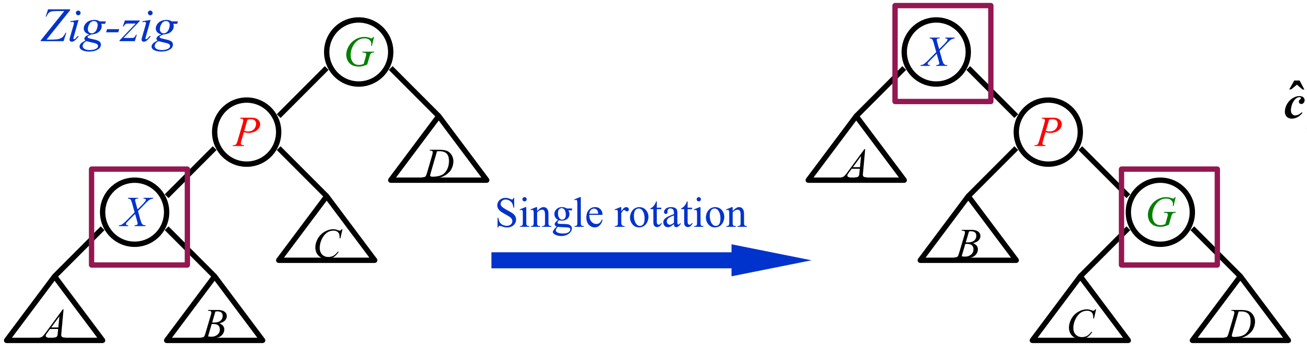


$$\hat{c}_i \le 1 + 3\big(R_2(X) - R_1(X)\big)$$

$$\begin{aligned}\hat{c}_i &= 2 + \cancel{R_2(X)} - \underline{R_1(X)} \\ &\quad + R_2(P) - \underline{R_1(P)} \\ &\quad + R_2(G) - \cancel{R_1(G)} \\ &\le 2\big(R_2(X) - \underline{R_1(X)}\big)\end{aligned}$$

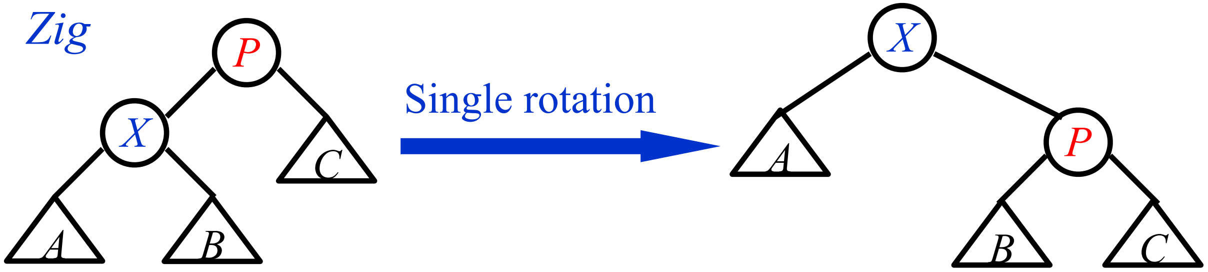
Lemma 11.4 on [Weiss] p.448

Zig-zig

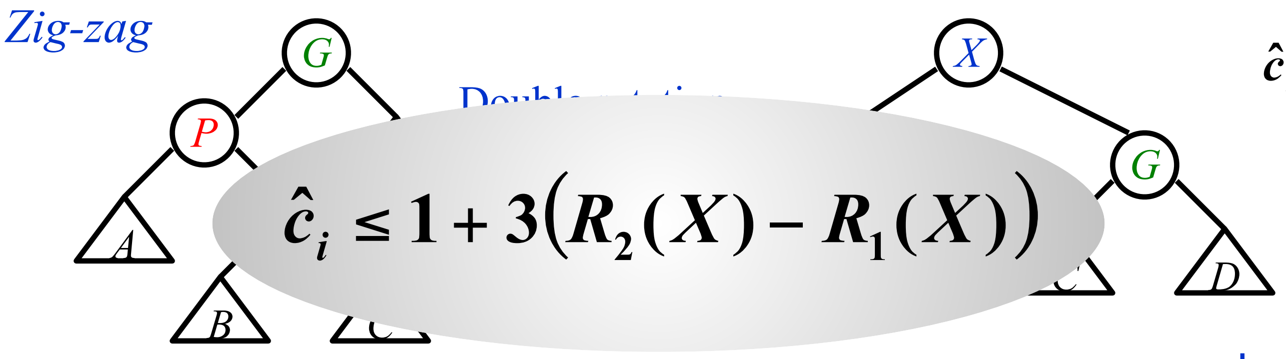


$$\begin{aligned}\hat{c}_i &= 2 + R_2(X) - R_1(X) \\ &\quad + R_2(P) - R_1(P) \\ &\quad + R_2(G) - R_1(G) \\ &\le 3\big(R_2(X) - R_1(X)\big)\end{aligned}$$

$$\Phi(T) = \sum_{i \in T} Rank(i)$$



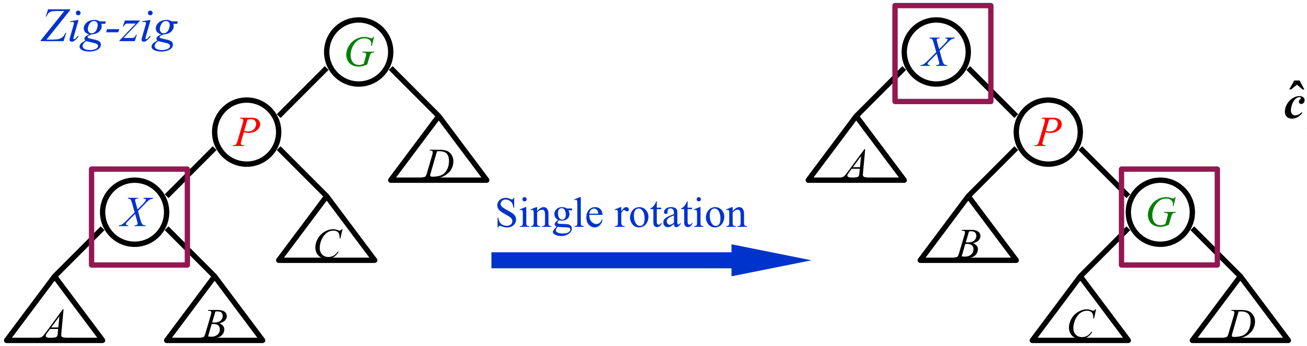
$$\hat{c}_i = 1 + R_2(X) - R_1(X) + \underline{R_2(P) - R_1(P)} \le 1 + R_2(X) - R_1(X)$$



$$\hat{c}_i \le 1 + 3\left(R_2(X) - R_1(X)\right)$$

$$\hat{c}_i = 2 + \cancel{R_2(X)} - \underline{R_1(X)} + R_2(P) - \underline{R_1(P)} + R_2(G) - \cancel{R_1(G)} \le 2\left(R_2(X) - \underline{R_1(X)}\right)$$

Lemma 11.4 on [Weiss] p.448



$$\hat{c}_i = 2 + R_2(X) - R_1(X) + R_2(P) - R_1(P) + R_2(G) - R_1(G) \le 3\left(R_2(X) - R_1(X)\right)$$

【Lemma】 The total cost of $\sum \hat{c}_i$ to splay a tree by a series of rotations with root *T* at node *X* is at most 3(*R*(*T*) – *R* (*X*)) + 1.

Amortized Cost of Splay Trees

【Lemma】 The total cost of $\sum \hat{c}_i$ to splay a tree by a series of rotations with root *T* at node *X* is at most 3(*R*(*T*) – *R* (*X*)) + 1.

Amortized Cost of Splay Trees

【Lemma】 The total cost of $\sum \hat{c}_i$ to splay a tree by a series of rotations with root *T* at node *X* is at most 3(*R*(*T*) – *R* (*X*)) + 1.

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1}) \right)$$

$$= \left(\sum_{i=1}^{n} c_i \right) + \underbrace{\Phi(D_n) - \Phi(D_0)}_{\geq 0}$$

Amortized Cost of Splay Trees

【Lemma】 The total cost of $\sum \hat{c}_i$ to splay a tree by a series of rotations with root *T* at node *X* is at most 3(*R*(*T*) – *R* (*X*)) + 1.

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1}) \right)$$

$$= \left(\sum_{i=1}^{n} c_i \right) + \underbrace{\Phi(D_n) - \Phi(D_0)}_{\geq 0}$$

Should assume to start from an empty tree

Amortized Cost of Splay Trees

【Lemma】 The total cost of $\sum \hat{c}_i$ to splay a tree by a series of rotations with root *T* at node *X* is at most 3(*R*(*T*) – *R* (*X*)) + 1.

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1}) \right)$$

$$= \left(\sum_{i=1}^{n} c_i \right) + \underbrace{\Phi(D_n) - \Phi(D_0)}_{\geq 0}$$

Should assume to start from an empty tree

We should also consider the influences of other steps other than rotations on the potential functions.
Fortunately, their influences are minor.

Amortized Cost of Splay Trees

【Lemma】 The total cost of $\sum \hat{c}_i$ to splay a tree by a series of rotations with root T at node X is at most $3(\underbrace{R(T)}_{\text{bounded by log(N)}} - R(X)) + 1$.

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right)$$

$$= \left(\sum_{i=1}^{n} c_i\right) + \underbrace{\Phi(D_n) - \Phi(D_0)}_{\geq 0}$$

Should assume to start from an empty tree

We should also consider the influences of other steps other than rotations on the potential functions.
Fortunately, their influences are minor.

Theorem:
The amortized cost of a series of operations started from an empty splay tree is of order O(log N), where N is the number of all nodes involved in the operations.
Read the original splay tree paper for details.

Balanced Binary Search Trees (1)

- Binary search trees
- AVL trees
- Splay trees
- Amortized analysis
- Take-home messages

Take-Home Messages

- Balanced binary search trees:
    - Reduce depth to reduce cost of operations.
- AVL trees:
    - Satisfying height-balanced condition. Conduct rotations to achieve self-balancing once the condition is violated.
- Splay trees:
    - Achieving self-balancing by conducting splaying steps for any operations.
- Amortized analysis:
    - Averaging the total cost which is limited by the structure.

Thanks for your attention!
Discussions?

Reference

Data Structure and Algorithm Analysis in C (2nd Edition)： Chap. 4.4-4.5, 11.5.

Introduction to Algorithms (4th Edition): Chap.16.

Daniel Dominic Sleator, Robert Endre Tarjan:
Self-Adjusting Binary Search Trees. Journal of ACM 32(3): 652-686 (1985)